DIGITAL RESEARCH™

CBASIC®
Language
Reference Manual

COPYRIGHT

Copyright © 1981, 1982 by Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

This manual is, however, tutorial in nature. Thus, the reader is granted permission to include the example programs, either in whole or in part, in his own programs.

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CBASIC, CP/M, CP/M-86, and CP/NET are registered trademarks of Digital Research. CBASIC-86, DDT, MP/M, and MP/M-86 are trademarks of Digital Research. Z80 is a registered trademark of Zilog Inc. Intel is a registered trademark of Intel Corporation.

The CBASIC Language Reference Manual was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press/ Monterey.

First Edition: November 1981
Second Edition: October 1982

Foreword

CBASIC® is a comprehensive and versatile programming language for developing professional microcomputer software. Software developers worldwide have selected CBASIC for its capacity to quickly produce reliable, maintainable programs in an enhanced programming environment. CBASIC combines the power of a structured, high-level language with the simplicity of BASIC to provide a serious development tool that is easy to learn and easy to use.

If you are a newcomer to data processing, read an introductory text on BASIC first. All you need is an understanding of elementary programming concepts and a familiarity with BASIC terminology to learn CBASIC.

The CBASIC *Language Reference Manual* covers CBASIC and CBASIC-86™.

- CBASIC runs under the CP/M®, MP/M™, and CP/NET® operating systems for computers based on the Intel® 8080, 8085 or the Zilog Z80® microprocessor.
- CBASIC-86 runs under the CP/M-86® or MP/M-86™ operating-systems for computers based on the Intel 8086 microprocessor.

Section 6 discusses the minor differences between the two versions of CBASIC.

At the end of Section 1 is a demonstration program that you can compile and run by following a few simple steps. The rest of the manual covers three main topics: CBASIC language definition, machine dependencies, and the Compiler and Interpreter.

- Sections 2, 3, and 4 define the CBASIC language.
- Section 5 covers input and output.
- Section 6 discusses assembly language interfacing and other machine-dependent topics.
- Section 7 discusses the Compiler, run-time Interpreter, and Cross-reference Lister.

Table of Contents

Appendixes

Tables

Figures

Section 1
Getting Started With CBASIC

1.1 CBASIC Components

The CBASIC system has two main components: the Compiler and the run-time Interpreter. CBASIC also provides a Cross-reference Lister.

- The CBASIC Compiler translates a source program into intermediate code. Source programs must be in .BAS files. The intermediate files are .INT files.
- The run-time Interpreter executes the .INT file that the Compiler generates.
- The Cross-reference Lister produces an alphabetized list of identifiers used in your CBASIC program. The Cross-reference Lister is a utility program provided as a convenience. It does not affect your programs.

1.2 Program Structure

CBASIC has features found in high-level languages, such as structured statements, functions, complex expressions, and data types. Some other CBASIC features are parameter passing, local and global variables, easy access to the operating system, and chaining between programs.

CBASIC requires no line numbers and allows the free use of commas, spaces, and tabs to make your programs more readable. A statement number or label is needed only when the statement is referenced from another location in the program. CBASIC allows integers, decimal fractions and exponential numbers as statement labels, as in the following examples:

```
1 PRINT “THESE ARE VALID LINE NUMBERS”

0 INPUT “ENTER A NUMBER:”;N

100 GOTO 100.0

100.0 END
```

```
21.543 A$ = NAME$

7920E12 Y = 2.0 * X
```

CBASIC statement labels do not have to be sequential. The Compiler treats the labels as strings of characters, not as numeric quantities. For example, the two labels 100 and 100.0 are distinct CBASIC statement labels. The maximum length for a statement label is 31 characters.

CBASIC statements can span more than one line. Use the backslash character, \, to continue a CBASIC statement on the next line. The Compiler ignores any character that follows a backslash on the same line. The backslash does not work as a continuation character if used within a string constant. The following example demonstrates the continuation character:

```
IF X = 3 THEN \
      PRINT “THE VALUES ARE EQUAL” \
ELSE \
      GOSUB 1000
```

In most cases, you can write multiple statements on the same line. Use a colon, :, to separate each command that appears on one line. However, the statements DIM, IF, DATA, and END cannot appear on one line with others. The following example demonstrates multiple statements on one line:

```
PRINT TAB(10);“X”: READ #1;NAME$: GOTO 1000
```

Use comments or remarks freely to document your programs. The REM statement allows unlimited program notation. Also, use spaces freely to enhance readability of your programs. Comments, long variable names, and blank spaces do not affect the size of your compiled program.

1.3 A Demonstration Program

The following demonstration program should help you get over the initial hurdle of compiling and running your first CBASIC program. You should already be familiar with CP/M and a text editor. The following instructions are for CBASIC on a CP/M-based system with two floppy-disk drives.

Make a back-up copy of your master CBASIC disk. Place your operating system disk into drive A and a copy of your CBASIC disk into drive B.

1. Write the program.

   Using your text editor, create a file named TEST.BAS on your CBASIC disk in drive B. Enter the following program into TEST.BAS exactly as it appears below:

```
PRINT
FOR I% = 1 TO 10
   PRINT I% "TESTING CBASIC!"
NEXT I%
PRINT
PRINT "FINISHED"
END
```

2. Compile the program.

   To start the CBASIC Compiler, enter the following command. Be sure drive B is the default drive.

```
B>CBAS TEST
```

   The Compiler assumes a filetype of .BAS for the file you specify in the Compiler command. A sign-on message, a listing of your source program, and several diagnostic messages display on your terminal. The message NO ERRORS DETECTED indicates a successful compilation. The Compiler creates an intermediate file for the TEST.BAS program. The directory for disk B should have the new file TEST.INT.

3. Run the program.

   To start the run-time Interpreter, enter the following command. Be sure drive B is the default drive.

```
B>CRUN TEST
```

The following output should appear on your terminal:

```
CRUN Ver. 2.XX Serial No.000-00000 Copyright (c)
1982 Digital Research, Inc. All rights reserved

1 TESTING CBASIC!
2 TESTING CBASIC!
3 TESTING CBASIC!
4 TESTING CBASIC!
5 TESTING CBASIC!
S TESTING CBASIC!
7 TESTING CBASIC!
8 TESTING CBASIC!
9 TESTING CBASIC!
10 TESTING CBASIC!

FINISHED
```

Minor differences appear in the sign-on message for the different versions of CBASIC.

End of Section 1

Section 2
Names, Numbers, and Expressions

CBASIC has three principal data types: integers, real numbers, and strings. CBASIC also supports dynamic, multidimensional arrays of all three data types. Each data type has a distinct form for identifiers. Numeric constants can be written in several forms.

CBASIC has a large set of operators for building expressions with variables, constants, and functions of the three data types. By converting from one type to another, where necessary, CBASIC allows real numbers to be mixed with integers in most expressions.

2.1 Identifiers

An identifier can be any length, as long as it fits on one line. Only the first thirty-one characters are meaningful for distinguishing one name from another. The first character must be a letter, the remaining characters can be letters, numerals, or periods. The final character in an identifier determines which data type it represents.

- Identifiers ending with $ are for strings.
- Identifiers ending with % are for integers.
- Identifiers without a $ or % are for real numbers.

The Compiler converts lower-case letters to upper-case letters unless toggle D is set.

Names for variables cannot begin with the letters FN. Names for user-defined functions always begin with FN.

The following are examples of valid CBASIC identifiers.

A%

NEW.SUM

```
file12.name$

Payroll.Identification.Number%
```

2.2 Strings

String constants are delimited by quotes. A string constant can have zero or more printing characters, as long as the string fits on a single line. The \ character has no special meaning inside a string constant. Two adjacent quotes represent one printed quote in the string.

For example, the string constant

```
“““Hello,”” said Tom.”
```

is stoned internally as the string:

```
“Hello ,” said Tom.
```

Although string constants cannot contain control characters and must fit on one line, string variables are more flexible. Internally, a string can have from 0 to 255 characters. Each character takes up one byte. The first byte in the string data area contains the length of the string. To build long strings, or to embed control characters in strings, use string expressions, as described later in this section, and string functions, as described in Section 3.

The following are examples of valid CBASIC string constants:

```
“July 4, 1776”

“Enter your name please:”

“““/”” has no special meaning inside a string.”
```

“” – the null string

2.3 Numbers

Two types of numeric quantities are supported by CBASIC: real and integer. A real constant is written in either fixed format or exponential notation. In both cases, it contains from one to fourteen digits, a sign, and a decimal point. In exponential notation, the exponent is of the form Esdd, where s, if present, is a valid sign, +, -, or blank, and where dd is one or two valid digits. The sign is the exponent sign and should not be confused with the optional sign of the mantissa. The numbers range from 1.0E-64 to 9.9999999999999E62. Although only fourteen significant digits are maintained internally by CBASIC, more digits can be included in a real constant. Real constants are rounded to fourteen significant digits.

A constant is treated as an integer if the constant does not contain an embedded decimal point, is not in exponential notation, and ranges from -32768 to +32767.

Integer constants can also be expressed as hexadecimal or binary constants. The letter H terminates a hexadecimal constant. The letter B terminates a binary constant. The first digit of a hexadecimal constant must be numeric. For example, 255 in hexadecimal is OFFH, not FFH. FFH is a valid identifier.

Hexadecimal and binary constants do not contain a decimal point. The value retained is the sixteen least-significant bits of the number specified.

In this manual, the terms real number and floating-point number are interchangeable. The term numeric applies to either a real or integer quantity.

Examples of valid numbers are:

```
1, 1.0, -99, 123456.789

1.993, .01, 4E12, 1.77E-9
```

1.5E+3 is equivalent to **1500.O**

1.5E-3 is equivalent to **.0015**

```
1ab0H, 10111110B, 0FFFFH
```

2.4 Variables and Array Variables

A variable in CBASIC represents an integer, real number, or a string, depending on the type of the identifier.

Each variable has a value always associated with it during program execution. A string variable does not have a fixed length associated with it. Rather, as different strings are assigned to the variable, the storage is dynamically allocated. The maximum length allowed in a string variable is 255 characters. Numeric variables are initialized to 0. String variables are initialized to the null string.

A variable takes the general form:

identifier [(subscript list)]

The following are examples of variables:

```
X$

PAYMENT

day.of.deposit%
```

Array variables look like regular variables with an added subscript list. CBASIC arrays can hold strings, integers, or reals. As with regular variables, the type of identifier specifies the type of array. The subscripts specify which element in the array to reference.

A subscript takes the general form:

expression {,expression}

The following examples show array variables:

```
y$(i%,j%,k%,l%)

COST(3,5)

POS%(XAXIS%,YAXIS%)

INCOME(AMT(CLIENT%),CURRENT.MONTH%)
```

The expressions in a subscript list must be numeric. Access to array elements is more efficient if integer expressions are used in subscript lists. If the expression is real, the value is rounded to the nearest integer before using the value. The subscript list indicates the variable is an array variable and indicates which element of the array reference.

When subscripts are calculated, a check ensures that the element selected resides in the referenced array. A run-time error occurs if the element does not reside in the referenced array. The run-time check ensures that the location calculated is included in the physical storage area of the array.

Before an array variable is referenced in a program, **it** must be dimensioned using the DIM statement. The DIM statement specifies the upper-bound of each subscript and allocates storage for the array. Section 3 describes the DIM statement.

An array must be dimensioned explicitly; no default options are provided. Arrays are stored in row-major order.

The subscript list is used to specify the number of dimensions and the extent of each dimension of the array being declared. The subscript list cannot contain a reference to the array being dimensioned. All subscripts have an implied lower-bound of zero.

The same identifier can name both a variable and an array variable in the same program, although that is not a recommended practice.

2.5 Expressions

The following are examples of expressions:

```
cost + overhead * percent
a*b/c(1.2+xyz)
last.name$ + ", " + first.name$
index% + 1
```

Expressions consist of algebraic combinations of function references, variables, constants and operators. Expressions evaluate to an integer, real, or string value. Table 2-1 gives the hierarchy of operators.

Table 2-1 Hierarchy of Operators

Hierarchy	*Operator*	*Definition*
1	( )	balanced parentheses
2	^	power operator
	Arithmetic Operators	
3	*, /	multiply, divide
4	+, -	plus, minus
	Relational Operators	
5	<	LT (less than)
	< =	LE (less than/equal to)
	>	GT (greater than)
	> =	GE (greater than/equal to)
	=	EQ (equal to)
	< >	NE (not equal)
	Logical Operators	
6	NOT	
7	AND	
8	OR	
9	XOR	

Arithmetic and relational operations work with integers and real numbers. An integer value converts to a real number if the operation combines a real and integer value. The operation then uses the two real values, resulting in a real value. This is mixed-mode arithmetic.

Mixed-mode operations require more time to execute because the Compiler generates more code. A mixed-mode expression always evaluates to a real value.

The power operator calculates the logarithm of the number being raised to the power if real values are used. A warning results when the number to the left of the operator is negative because the logarithm of a negative number is undefined. The absolute value of the negative quantity calculates the result. The exponent is positive or negative.

If both values used with the power operator are either integer constants or integer variables, the result is calculated by successive multiplication. This allows a negative integer number to be raised to an integer power. With integers, if the exponent is negative, the result is zero. In all cases, 0 ^ 0 is 1 and 0 ^ X, where X is not equal to 0, is 0.

If the exponent is an integer but the base is real, the integer is converted to a real value before calculating the result. Likewise, if the exponent is real but the base is an integer quantity, the result is calculated using real values.

String variables can only be operated on by the relational operators and the concatenation operator +. Mixed string and numeric operations are not permitted. The mnemonic relational operators (LT, LE, etc.) are interchangeable with the corresponding algebraic operators (<, <= etc.).

Relational operators result in integer values. A 0 is false and a -1 is true. Logical operators AND, NOT, OR, and XOR operate on integer values and result in an integer number. If a real value is used with logical operators, it is first converted to an integer.

If a numeric quantity exceeds the range from 32,767 to -32,768, it cannot be represented by a 16-bit two's complement binary number. Logical operations on such a number produce unpredictable results.

These are results of logical operations:

`12 AND 3`	= 0	`1100B AND 0101B`	= 4
`NOT -1`	= 0	`NOT 3H`	= -4
`12 OR 3`	= 15	`0CH OR 5H`	= 13
`12.4 XOR 3.2`	= 15	`12.4 XOR 3.7`	= 8

Efficiency is increased by using integer expressions for relational tests and logical operations. Programs written in Version 1 of CBASIC should be converted to use integer variables wherever possible.

Note: if a series of digits contains no decimal point or ends in a decimal point, CBASIC attempts to store it as an integer. If the resulting number is in the range of CBASIC

integers, it is treated as an integer. If the constant is then required in an expression as a real number, the constant converts to a real number at run-time. For example,

```
X = X + 1.
```

causes the integer constant 1. to be converted to a real value before adding it to X. To eliminate this extra conversion, embed the decimal in the number as shown below:

```
X = X + 1.0
```

Actually, there is very little difference in execution speed. A similar situation exists in the following statement:

```
Y% = X% + 1.0
```

In this case, the X% is converted to a real number before adding it to the real constant. The result is then converted back to an integer prior to assignment to Y%.

Generally you should avoid mixed-mode expressions whenever possible, and not use real constants with integer variables. Most whole numbers used in a program are stored as integers. This provides the most effective execution.

If an overflow occurs during an operation between real values, a warning is printed. Execution continues, with the result of the operation set to the largest real number.

In the case of integers, no checking for overflow is performed because this reduces the efficiency of integer operations. The calculated value returns a negative number if the results of an integer operation fall outside the range of integer values, greater than 32767 and less than -32767.

End of Section 2

Section 3
Statements and Functions

This section uses the following typographical conventions to highlight the various elements that make up each statement and function.

- CAPS indicate CBASIC keywords.
- Lowercase letters identify variables.
- Italics indicate syntactic items, such as expressions.
- Items enclosed in square brackets [ ] are optional.
- Items enclosed in braces { } are optional and can be repeated.

ABS Function

The ABS(x) function returns the absolute value of the expression x.

Syntax:

y = ABS(*numeric expression*)

Explanation:

The ABS function returns a floating point number. If the numeric expression is an integer, CBASIC converts it to floating point. If the expression is positive or negative, the function returns a positive value. If the expression is zero, the function returns zero.

Examples:

```
DISTANCE = ABS(START - FINISH)
IF ABS(DELTA.X) <= LIM THEN STOP
```

ASC Function

The ASC(a$) function returns the ASCII decimal value of the first character in the string argument.

Syntax:

x% = ASC(*string expression*)

Explanation:

The function argument must be a string that is at least one character long, otherwise the function produces a run-time error.

See also the CHR$ function, which is the inverse of ASC.

Examples:

```
IF ASC(DIGIT$)>47 AND ASC(DIGIT$)<58 THEN
      PRINT "VALID DIGIT"

OUT TAPE.PORT%, ASC("*")
```

ATN Function

The ATN(x) function returns the arc tangent of x.

Syntax:

y = ATN(*numeric expression*)

Explanation:

The result of ATN(x) is the angle, expressed in radians, whose tangent is x. The result is a floating-point number.

Examples:

```
RADIANS = ATN(X)

IF ATN(N) < PI/2.0 THEN\

PRINT “ANGLE LESS THAN SO DEGREES”
```

CALL Statement

The CALL statement links to a machine language subroutine.

Syntax:

CALL <*numeric expression*>

Explanation:

The statement calls the machine language subroutine address specified by the expression. If the value is a real number, CBASIC rounds it to the nearest integer.

Section 6 discusses machine language interfacing and addressing on different microprocessors.

Examples:

See Section 6.5 for examples of the CALL statement.

CHAIN Statement

The CHAIN statement transfers control to another program.

Syntax:

CHAIN *filespec*

Explanation:

The CHAIN statement transfers control to the program specified in the filespec (file specification) . A drive specification is optional; the default is the currently logged-in drive. The file you specify must be of type INT. Even if you specify a different file type, the statement only recognizes type INT files. See Section 7.4 for additional information.

Examples:

```
CHAIN “B:PAYROLL”
CHAIN SEGMENT$
```

CHR$ Function

The CHR$(x) function returns the character whose ASCII decimal value is x.

Syntax:

a$ = CHR$(*numeric expression*)

Explanation:

The result of CHR$ is a single-character string, whose ASCII value is equal to the value of the input expression. If the expression is in floating point, CBASIC converts it to an integer.

Appendix D lists the character set and their ASCII values. Use CHR$ to send control characters to an output device, as shown in the examples below.

Examples:

```
PRINT CHR$(7)     REM BEEP THE TERMINAL

PRINT CHR$(LINEFEED%)

IF CHR$(INP(IN.PORT%)) = "A" THEN GOSUB 100
```

CLOSE Statement

The CLOSE statement deactivates open files.

Syntax:

CLOSE *file number* {,*file number*}

Explanation:

The CLOSE statement deactivates an open file. This means that the file is no longer available for input or output. The specified file must have been activated by a CREATE, FILE, or OPEN statement before using CLOSE.

Each expression refers to the identification number of an active file. The expression must be an integer ranging from 1 to 20. CBASIC converts real numbers to integers.

The CLOSE statement closes the file, releases the file number, and reallocates all buffer space used by the file. IF END statements assigned to closed files have no further effect.

A STOP statement automatically closes all active files. A CTRL-Z entered in response to an INPUT statement closes all active files. A CTRL-C does not close active files. Run-time errors do not close active files.

Examples:

```
CLOSE FILE.NO%

CLOSE NEW.MASTER.FILE%, OLD.MASTER.FILE%, UPDATE.812%

FOR X% = 1 TO NO.OF.WORK.FILES%
      CLOSE X%
NEXT X%
```

COMMAND$ Function

The COMMAND$ function returns a string containing the parameters from the command line that started the program.

Syntax:

a$ = COMMAND$

Explanation:

COMMAND$ returns the command line from the operating system, minus the program name. The word TRACE and any associated line numbers are not included in the string if the TRACE option is in the command line.

COMMAND$ removes leading blanks and converts alphabetic characters to uppercase. The maximum length of the returned string is 50 characters.

Use the COMMAND$ function anywhere in the program, any number of times, and with any program loaded by a CHAIN statement.

Section 7 discusses command lines in more detail.

Examples:

```
IF COMMAND$ = "" THEN STOP
```

If any of the following commands starts a CBASIC program,

```
CRUN PAYROLL NOCHECKS TOTALS
```

```
CRUN86 payroll nochecks totals
```

```
CRUN86 payroll trace nochecks totals
```

the resulting string from COMMAND$ is

```
NOCHECKS TOTALS
```

COMMON Statement

The COMMON statement specifies variables that are common to the main program and all programs executed through CHAIN statements.

Syntax:

COMMON *variable* {,*variable*}

Explanation:

A COMMON statement is a non-executable statement that specifies listed variables that are common to the main program, and all programs executed through a CHAIN statement. If present, COMMON statements must be the first statements in a program. However, blank lines and REM statements can precede COMMON statements.

If the main program contains COMMON statements, each chained program must have COMMON statements that match those in the main program. Matching means that there are the same number of variables in each COMMON statement, and that the type of each variable in the COMMON statement of the main program matches the type of each variable in the COMMON statement of the chained program. Also, dimensioned variables must have the same number of subscripts in each program.

Specify array variables by placing the number of subscripts in parentheses after the array name as shown in the following example.

Examples:

To declare an array such as ARRAY(S1, S2, S3) in a COMMON statement, place the number of subscripts in parentheses after the array name as shown in the following COMMON statement.

```
COMMON ARRAY (3)
```

The next COMMON statement specifies X and Y as non-subscripted real variables, common to all chained programs. A and B$ are arrays common to all chained programs. A has three subscripts, while B$ has two. The COMMON statement does not indicate the size of an array subscript.

```
COMMON X, Y, A(3), B$(2)
```

Note: an array must be declared with a DIM statement before you can access an element in the array

The first program that requires access to the array must contain a DIM statement that specifies the desired range for each subscript. Subsequent programs can access this array through COMMON statements in the chaining process. The data in the array remains unchanged. However, if a subsequent program executes another DIM statement for this array, the data in the array is lost. In other words, the array is reinitialized.

Elements in string arrays are not released from memory during re-initialization. You can set the elements of a string array to null strings before executing a second DIM statement for that array.

CONCHAR% Function

CONCHAR% reads one character from the console device.

Syntax:

i% = CONCHAR%

Explanation:

The value returned is an integer. The low-order eight bits of the returned value are the binary representation of the ASCII character input. The high-order eight bits are zero.

Examples:

```
i% = CONCHAR%

CHAR% = 0

IF CONSTAT% THEN\
      CHAR% = CONCHAR%

IF CHAR% = STOPCHAR% THEN\
      RETURN
```

CONSOLE Statement

The CONSOLE statement restores printed output to the console.

Syntax:

CONSOLE

Explanation:

The console is the physical unit currently assigned to CON: by CP/M.

If the list device print position is not 1, a carriage return and line-feed are output to the list device.

The width of the console device is changed with the POKE statement. The console width is one byte at location 272 base 10, or 110H. The new console width becomes effective at the next execution of a CONSOLE statement. The console line width is initially set to 80, 50H.

A width of zero results in an infinite width. With a zero width in effect, carriage returns and line-feeds are never automatically output to the console as a result of exceeding the line width.

Examples:

```
490 CONSOLE

IF END.OF.PAGE% THEN\
CONSOLE:\
PRINT USING "##,### WORDS THIS PAGE";WORDS%:\
INPUT "INSERT NEW PAGE, THEN CR";LINE NULL.STRING$
LPRINTER
```

CONSTAT% Function

CONSTAT% returns the console status as a boolean integer value.

Syntax:

i% = CONSTAT%

Explanation:

If the console device is ready, a logical true, -1, is returned, otherwise a logical false, 0, is returned.

Examples:

```
IF CONSTAT% THEN\
      GOSUB 100 REM PROCESS OPERATOR INTERRUPT

WHILE NOT CONSTAT%
      WEND
```

COS Function

COS(x) returns the cosine of x.

Syntax:

y = COS (x)

Explanation:

The argument x is expressed in radians. The value returned by COS is real. If x is an integer, CBASIC converts it to a real number.

Examples:

```
IF COS(ANGLE) = 0.0 THEN VERTICAL% = TRUE%

PRINT CONSTANT * COS(ROTATION)
```

CREATE Statement

The CREATE statement is identical to an OPEN statement except that a new file is created on the selected drive.

Syntax:

CREATE *filespec* {RECL *reclength*}\
AS *file number* {BUFF *number of sectors*\
RECS *size of sector*}

Explanation:

When a file with the same name is present, the existing file is erased before the new file is created.

The CREATE statement has no effect on any IF END statement currently in effect for the identification number assigned to the new file.

Examples:

```
1200 CREATE "NEW.FIL" AS 19 BUFF 4 RECS 128

CREATE ACC,MASTER$ RECL M.REC.LEN% AS ACC.FILE.NO%

CREATE "B:" + NAME$ + "." + LEFT$(STR$(CURRENT.WORK%),3)
      AS CURRENT.WORK%
```

DATA Statement

DATA statements are non-executable statements that define string, real, and integer constants assigned to variables using the READ statement. Any number of DATA statements can occur anywhere in a program.

Syntax:

DATA *constant* {, *constant*}

Explanation:

The constants are stored consecutively in a data area as they appear in the program and are not syntax checked by the Compiler. Strings can be enclosed in quotation marks or optionally delimited by commas.

A DATA statement must be the only statement on a line; it cannot be continued with a continuation character. However, all DATA statements in a program are treated collectively as a concatenated list of constants separated by commas.

If a real constant is assigned to an integer variable with a READ statement, the constant is rounded off to the integer portion of the real number. If the value of a number assigned to an integer is outside the range of CBASIC integers, incorrect values are assigned. If a real number exceeds the range of real numbers, an overflow warning occurs and the largest CBASIC number is used in its place.

Examples:

```
400 DATA 332.33, 43.0089E5, "ALGORITHM"

DATA ONE, TWO, THREE, 4, 5, 6
```

DEF Statement

Use the DEF statement to define both single- and multiple-statement functions.

Syntax:

Single: DEF FN *function.name*[(dummy arg list)] = expression

Multiple: DEF FN *function.name*[(dummy arg list)] CBASIC statements
.
.
.
RETURN
FEND

Explanation:

A function definition must occur in a program before making a function reference. To define a function, the keyword DEF must precede the function name. CBASIC supports two types of function definitions: single-statement and multiple-statement. Refer to Section 4 for more information.

Examples:

Single:
```
DEF FN.CALC = RND *25.0
```

Multiple:
```
DEF FN.WRITE OUTPUT (OUTPUT.NO%)
      PRINT# OUTPUT.NO%; CUSTNO%,AMOUNT
      RETURN
FEND
```

DELETE Statement

The DELETE statement removes the referenced files from their respective directories.

Syntax:

DELETE *file number* {,*file number*}

Explanation:

Each expression must be in the range of 1 to 20. If the number is not currently assigned to an active file, a run-time error occurs. The expression must be numeric. Real numbers are converted to integers. A string value results in an error.

If an IF END statement is currently associated with the identification number for the file being deleted, the IF END is no longer in effect.

Examples:

```
DELETE 1

DELETE FILE,NO%, OUTPUT.FILE.NO%

I% = 0
WHILE I% < NO.OF.WORKFILES%
      I% = I% + 1
      DELETE I%
WEND
```

DIM Statement

The DIM statement dynamically allocates space for an array.

Syntax:

DIM *identifier* (*subscript list*)
{,i*dentifier* (*subscript list*)}

Explanation:

A DIM statement is an executable statement; each execution allocates a new array. Before CBASIC references an array variable in a program, the variable must be dimensioned using the DIM statement. The DIM statement specifies the upper-bound of each subscript and allocates storage for the array.

The DIM statement dynamically allocates space for numeric or string arrays. If the array contains numeric data, the previous array is deleted before allocating space for a new array. If the array is string, each element must be set to a null string before re-executing the DIM statement to regain the maximum amount of storage.

Elements of string arrays are any length up to 255 bytes, and change in length as different values are assumed. Initially, numeric arrays are set to zero and all elements of string arrays are null strings.

Examples:

```
DIM A(l0)

DIM ACCOUNTS$(100),ADDRESS$(100),NAME$(100)

DIM B%(2,5,10), SALES.PERSON%(STAFF.SIZE%)

DIM X(A%(I%),M%,N%)
```

END Statement

The END statement terminates a source program.

Syntax:

END

Explanation:

The END statement is a directive to the Compiler indicating an end to the source program. The Compiler ignores any statements that follow an END statement.

An END statement cannot appear on the same line with other statements.

Examples:

```
500 END
```

```
END
```

EXP Function

EXP returns the value of the irrational constant e, raised to the power given by x.

Syntax:

y = EXP(x)

Explanation:

The value returned by EXP is real. If x is an integer, CBASIC converts it to a real number.

Examples:

```
Y = A * EXP(BX%)

E=EXP(1)    REM CONSTANT E = 2.7182.....
```

FEND Statement

FEND is the multiple-line, user-defined function terminator statement.

Syntax:

FEND

Explanation:

FEND must appear only once, at the end of the function definition.

The run-time Interpreter detects an error condition and aborts the program if the interpreter executes a FEND statement. Therefore, always place a RETURN statement in a function definition.

Examples:

```
350 FEND
```

```
FEND
```

FILE Statement

A FILE statement opens an existing file for reading or updating. If the file does not exist, the FILE statement creates it.

Syntax:

FILE *flespec* [(*rec length*)]
{, *filespec* [(*rec length*)]}

Explanation:

The filespec contains the name of the file to be accessed. As each file is activated, the file is assigned the next unused file number starting with 1. If all 20 numbers are assigned, an error occurs.

The record length must be a numeric expression. Real numbers are converted to integers. A string value causes an error.

The variable must not be subscripted, and it must be string. It cannot be literal or an expression.

Examples:

```
FILE NAME$

FILE FILE.NAME$(REC.LEN%)
```

FLOAT Function

FLOAT(i%) converts the argument i% to a real value.

Syntax:

y = FLOAT(i%)

Explanation:

The argument should be numeric. If i% is real, CBASIC first converts it to an integer, and then back to a real number.

Examples:

```
AMOUNT = FLOAT(COST%)

POSITION = SIN(FLOAT(ANG%)) * OFFSET
```

FOR Statement

The FOR statement controls a FOR/NEXT loop.

Syntax:

FOR *index* = numeric exp. TO
numeric exp [STEP *numeric exp.]*

Explanation:

Execution of all statements between the FOR statement and its corresponding NEXT statement repeats until the indexing variable, incremented by the STEP expression after each iteration, reaches the exit criteria.

If the STEP expression is positive, the loop exit criteria is met when the index exceeds the value of the TO expression. If the STEP expression is negative, the index must be less than the value of the TO expression for the exit criteria to be satisfied.

The index cannot be an array variable initially set to the value of the first expression. Both the TO and STEP expressions are evaluated on each loop; all variables associated with these expressions can change in the loop.

Also, the index can be changed during execution of the loop. The type of the index can be real or integer, but all expressions must be the same. If any of the expressions are string, an error occurs. Particular care should be taken to ensure proper matching of the expression types. For example,

```
FOR I% = 1 TO DONE
```

generates unnecessary code because DONE is real, but 1% and 1 are integers. Here is a more subtle example,

```
FOR I = 1. TO DONE
```

where I and DONE are real, but 1. is an integer.

There is one situation when a FOR statement, which appears valid, generates Compiler error FE. This occurs if the type of the expression following the TO is not the same as the type of the loop index variable.

For example,

```
FOR I = 1 TO 13 STEP 3
```

results in an FE error because the index variable I is real, but the value following the TO is an integer. Changing the index to I% eliminates the error.

If you omit the STEP clause, a default value of one is assumed. The type of the STEP expression in this case is the same as the type of the index.

The statements within a FOR loop are always executed at least once.

If you want a step of one, omit the STEP clause. The execution is much faster because fewer run-time checks are made. Also, less intermediate code is produced. Execution speed also substantially improves if all the expressions are integer.

Examples:

```
FOR INDEX% = 1 TO 10
      SUM = SUM + VECTOR(INDEX%)
NEXT INDEX%

FOR POSITION=MARGIN+TABS TO PAPER.WIDTH STEP TABS
      PRINT TAB(POSITION);SET.TAB$;
NEXT POSITION
```

Note: In the CP/M-86 implementation of CBASIC, the upper limit for an integer index variable in a FOR loop is 32766. A larger value causes an infinite loop.

FRE Function

FRE returns the number of bytes of unused space in the free storage area.

Syntax:

y = FRE

Explanation:

The value returned by FRE is a floating-point number.

Examples:

```
X = FRE

IF FRE < 500.0 THEN GOSUB 10 REM PRINT WARNING
```

GOSUB Statement

The GOSUB statement transfers statement execution to a statement specified by a reference to a label.

Syntax:

GOSUB *stint number*

GO SUB *stmt number*

Explanation:

In a GOSUB or GO SUB statement, CBA SIC saves the location of the next sequential instruction on the return stack. Control then transfers to the statement labeled with the statement number following the GOSUB. A subroutine call can be nested 20 deep. See the RETURN statement for more information.

Examples:

```
GOSUB 700

PRINT “BEFORE TABLE”
GOSUB 200   REM PRINT THE TABLE
PRINT “AFTER TABLE”
STOP
200   REM PRINT THE TABLE
      FOR INDEX% = 1 TO TABLE.SIZE%
            PRINT TABLE(INDEX%)
      NEXT INDEX%
      RETURN
```

GOTO Statement

The GOTO statement transfers execution to a statement identified by a statement number.

Syntax:

GOTO *stmt number*

GO TO *stmt number*

Explanation:

Following a GOTO or GO TO statement, execution continues at the statement labeled with the statement number. If the statement number branched to is not an executable statement, execution continues with the next executable statement after the statement number.

If control is transferred to a non-existing statement number, an error occurs.

Examples:

```
80 GO TO 35

GOTO 100.5
```

IF Statement

Syntax:

IF *variable* THEN *statement list*
[ELSE *statement list*]

IF *variable* THEN *stmt number*

Explanation:

An IF statement expression is usually a logical expression evaluating to either true (-1) or false (0). However, CBASIC accepts any numeric expression of type integer, treating a value other than zero as true. This reduces both execution time and intermediate file size generated by the Compiler. If the expression is real, the value is rounded and converted to an integer. A string expression results in an error.

A statement list is composed of one or more statements where a colon separates each pair of statements. The colon is not required after the THEN, nor is it required before or after the ELSE; it only separates statements.

An IF statement must be the first statement on a line; it cannot follow a colon. Therefore, IF statements cannot be nested.

Examples:

```
IF ANSWER$="YES" THEN GOSUB 500

IF DIMENSIONS.WANTED% THEN PRINT LENGTH, HEIGHT

IF VALID% THEN \
      PRINT MSG$(CURRENT.MSG%) :\
      GOSUB 200 :\      UPDATE RECORD
      GOSUB 210 :\      WRITE RECORD
      NO.OF.RECORDS%=NO.OF.RECORDS%+1 :\
RETURN

IF X > 3 THEN X = 0 : Y = 0 : Z = 0

IF YES% = TRUE% THEN PRINT MSG$(1) \
            ELSE PRINT MSG$(2)
```

```
IF TIME > LIMIT THEN \
      PRINT TIME.OUT,MSG$ :\
      BAD.RESPONSES% = BAD.RESPONSES%+1 :\
      QUESTION% = QUESTION% + 1 \
ELSE \
      PRINT THANKS.MSG$ :\
      GOSUB 1000 :\ ANALYSE RESPONSE
      ON RESPONSE% GOSUB \
            2000, 2010, 2020, 2030, 2040 :\
      RETURN
```

In the preceding examples, the colon separates statements in a statement list, and the backslash continues a statement onto another line.

Because the Compiler ignores anything on or following the same line with the backslash, comments can be inserted without using the keyword REM.

If the value of the expression is not zero, the statements in the first statement list are executed. Otherwise, the statement list following ELSE is executed, if present, or the next sequential statement following the IF statement is executed.

In the second example of the IF statement, when the expression is not equal to zero, an unconditional branch to the statement number occurs. This form of the IF statement does not have an ELSE clause. This variation is included in CBASIC for compatibility with previous versions of Basic.

IF END Statement

The IF END statement allows you to process an end-of-file condition on an active file.

Syntax:

IF END # *file number* THEN *stmt number*

Explanation:

When an end-of-file is detected, one of two actions takes place. If an IF END statement is executed for the file, control transfers to the statement labeled with the statement number following THEN. If an IF END statement was not executed, a runtime error occurs.

The IF END statement must be the only statement on a line; it cannot follow a colon or be part of a statement list.

Any number of IF END statements can appear in a program for a given file. The most recently executed IF END is the one in effect. However, if a DELETE or CLOSE statement is executed, any IF END statement associated with the identification number is no longer effective.

The file number must be in the range of 1 to 20. Real numbers are converted to integers.

When a condition causes the transfer of control to the statement associated with an IF END statement, the stack is restored to the condition that existed before the statement causing activation of the IF END statement.

Thus if the statement that resulted in transfer was in a subroutine, a return must be executed after processing the end-of-file condition.

An IF END statement can be executed before assigning the file number to a file. A subsequent OPEN on a file that does not exist causes execution to continue as if an end-of-file were encountered.

In the following example, if the file MASTER.DAT does not exist on drive B, control transfers to statement 500.5. After a successful OPEN, an end-of-file during a READ continues execution with statement 500:

```
IF END #MASTER.FILE.NO% THEN 500.5
OPEN "B:MASTER.DAT" AS MASTER.FILE.NO% BUFF 6 RECS 128
IF END # MASTER.FILE.NO% THEN 500
```

An IF END statement can also be used when writing to a file. In this case, control transfers to the statement associated with the IF END when an attempt is made to write to the file and there is no disk space available. Part of the record created is written to the file. When using fixed files, the last record is rewritten after more space is freed.

Examples:

```
IF END # 7 THEN 500

IF END # FILE.NO% THEN 100.1
```

INITIALIZE Statement

The INITIALIZE statement allows the changing of diskettes during program execution without restarting the operating system.

Syntax:

INITIALIZE

Explanation:

INITIALIZE must execute after making the diskette change. Be sure never to change diskettes while any files are open.

Examples:

```
350 INITIALIZE

INITIALIZE
```

INP Function

INP returns the value input from the CPU I/O port specified. This function is useful for accessing peripheral devices directly from the CBASIC program.

Syntax:

y% = INP (*numeric exp.*)

Explanation:

The argument must be numeric. For the results to be meaningful, the argument must range from 0 to 255 for the 8-bit version, and from 0 to 65535 for the 16-bit version. If it is a string, an error occurs. A real value is rounded to the nearest integer.

Examples:

```
PRINT INP(ADDR%)

IF INP(255) > 0 THEN PRINT CHR$(7)

ON INP(INPUT.DEVICE.PORT%) GOSUB \
      100, 200, 300, 400, 400, 400, 500
```

INPUT Statement

INPUT statements accept data from the console and assign the data to program variables.

Syntax:

INPUT [*prompt string* ;]
variable {, *variable*}

Explanation:

If a prompt string is present in an INPUT statement, CBASIC prints it on the console; otherwise, a ? is output. In both cases, a blank is then printed and a line of input data is read from the console and assigned to the variables as they appear in the variable list.

The variables can be either simple or subscripted string, or numeric.

The maximum number of characters that can be entered in response to an INPUT statement is 255. If 255 or more characters are entered, inputting automatically ends and the first 255 characters are retained. Additional characters are lost. The 255 characters include all characters entered in response to an INPUT statement, no matter how many variables appear in the variable list.

All CP/M line editing functions, such as CTRL-U and DELETE, are in effect. A CTRL-C terminates the program without closing open files. If a CTRL-Z is the first character entered in response to an INPUT statement, the program ends in the same manner as if a STOP statement was executed.

The data items entered at the console must be separated by commas and are ended by a carriage return. Strings enclosed in quotation marks allow commas and leading blanks to be included in the string.

The prompt string must be a string constant. If it is an expression or a numeric constant, an error occurs.

If the value entered for assignment to an integer is real, the number entered is truncated to the integer portion of the real number, If the value of a number assigned to an integer variable is outside the range of integers, an incorrect value is assigned.

If a real number exceeds the range of CBASIC real numbers, the largest real number is assigned to the variable, and a warning is printed on the console.

If too many or too few data items are entered, a warning is printed on the console, and the entire line must be reentered.

Examples:

```
INPUT AMOUNT 1, AMOUNT2, AMOUNT3

INPUT "WHAT FILE, PLEASE?";FILE.NAME$

INPUT "YOUR PHONE NUMBER PLEASE:"; PHONE.N$

INPUT"";ZIP.CODE%
```

A special type of INPUT statement is the INPUT LINE. The general form of this statement is:

INPUT [*prompt string;*]
LINE *variable*

Some examples are:

```
INPUT "ENTER ADDRESS";LINE ADDR$

INPUT "TYPE RETURN TO CONTINUE";LINE DUMMY$
```

The INPUT LINE statement functions as described above with the following exception: only one variable is permitted following the keyword LINE. It must be string. Any data entered from the console is accepted and assigned to this variable. The data is terminated by a carriage return.

A null string is accepted by responding to a INPUT LINE statement with a carriage return. If the variable specified to receive the input is not string, an error occurs.

Prompt strings are directed to the console even when a LPRINTER statement is in effect (See Section 4.3).

INT Function

INT(x) returns the integer part of the argument x; the fractional part is truncated.

Syntax:

y = INT(x)

Explanation:

The value returned by INT is a floating-point number. The argument should be numeric. If x is an integer, CBASIC converts it to a real number.

Examples:

```
TIME=INT(MINUTES)+INT(SECONDS)

IF (X/2)-INT(X/2)=0 THEN PRINT \
      "EVEN" ELSE PRINT "ODD"
```

INT% Function

INT% (x) converts the argument x to an integer value.

Syntax:

y% = INT%(x)

Explanation:

The argument should be numeric. If x is an integer, it is first converted to a real number, and then converted back to an integer.

Examples:

```
J% = INT%(REC.NO)

WIDTH% = DIMEN.1% + INT%(DIMEN.2)
```

LEFT$ Function

LEFT$ returns a string consisting of the first i% characters of a$.

Syntax:

b$ = LEFT$(a$, i%)

Explanation:

If i% is greater than the length of a$, LEFT$ returns the entire string. If i% is zero, a null string is returned. If i% is negative, a run-time error occurs.

a$ must be a string; otherwise, an error occurs. i% should be numeric. If i% is real, CBASIC converts it to an integer. If i% is a string, an error occurs.

Examples:

```
PRINT LEFT$(INPUT.DATA$,25)

IF LEFT$(IN$,1) = "Y" THEN GOSUB 400
```

LEN Function

LEN returns the length of a$.

Syntax:

i% = LEN(a$)

Explanation:

If a$ is a null string, LEN returns zero.

The value returned by LEN is an integer. If the argument is numeric, an error occurs.

Examples:

```
IF LEN(TEMPORARY$) > 25 THEN \
      TOO.LONG% = TRUE%

FOR INDEX% = 1 TO LEN(OBJECT$)
      NUM%(INDEX%) = ASC(MID$(OBJECT$,INDEX%,1))
NEXT INDEX%
```

LET Statement

The LET statement assigns a value to a variable.

Syntax:

[LET] variable = expression

Explanation:

The expression is evaluated and assigned to the variable appearing on the left side of the =. The variable and expression must both be string or numeric type.

If the variable and expression are both numeric but one is integer and the other is real, an automatic conversion to the type of the variable on the left of the = is performed.

Examples:

```
100 LET A = B + C

X(3,POINTER%) = 7.32 * Y + X(2,3)

SALARY = (HOURS.WORKED * RATE) - DEDUCTIONS

date$ = month$ + " " + day$ + ", " + year$

INDEX% = INDEX% + 1

REC.NUMBER = OFFSET% + NEXTREC%
```

LOG Function

LOG(x) returns the natural logarithm of x.

Syntax:

y = LOG(x)

Explanation:

The natural or Naperian logarithm of the argument x is the Base e inverse of the EXP function.

The value returned by LOG is real. If x is an integer, CBASIC converts it to a real number.

Examples:

```
BASE.TEN.LOG = LOG(X)/LOG(10)

PRINT “LOG OF X IS ”; LOG(X)
```

LPRINTER Statement

[PRINTER sends output to the printer or list device.

Syntax:

LPRINTER [WIDTH *numeric exp.*]

Explanation:

After execution of the LPRINTER statement, all PRINT statement output, usually directed to the console, is output onto the list device.

The list device is the physical unit currently assigned to LST: by CP/M. The WIDTH clause is optional. If present, the expression is used to set the line width of the list device.

If the console cursor position is not 1, a carriage return and line-feed is output to the console. In this context, the cursor position is the value returned by the POS function before executing the LPRINTER statement.

The expression should return an integer. If it is real, the value is rounded to an integer. If the expression is string, an error occurs.

If the WIDTH clause option is not present, the most recently assigned width is used. Initially the width is set to 132. A width of 0 results in an infinite line width. With a zero width in effect, carriage returns and line-feeds are never automatically output to the printer as a result of exceeding the line width.

Examples:

```
500 LPRINTER

IF HARDCOPY,WANTED% THEN LPRINTER WIDTH 120

LPRINTER WIDTH REQUESTED.WIDTH%
```

MATCH Function

MATCH returns the position of the first occurrence of a$ in b$ starting with the character position given by the third parameter. A zero is returned if no MATCH is found.

Syntax:

j% = MATCH(a$, b$, i%)

Explanation:

The following pattern-matching features are available:

- # matches any digit (0-9).
- ! matches any upper- or lower-case letter.
- ? matches any character.
- \ serves as an escape character indicating the following character does not have special meaning. For example, a ? signifies any character is a MATCH unless preceded by a \.

a$ and b$ must be strings. If either of these arguments are numeric, an error occurs. If i% is real, it is converted to an integer. If i% is a string, an error occurs. If i% is negative or zero, a run-time error occurs. When i% is greater than the length of b$, zero is returned. If b$ is a null string, a 0 is returned. If b$ is not null, but a$ is null, a 1 is returned.

The following program experiments with the MATCH function:

```
TRUE% = -1
FALSE% = 0
edit$ = " The number of occurrences is ###"
WHILE TRUE%
      INPUT "enter object string" ; LINE cbject$
      INPUT "enter argument string" ; LINE arg$
      GOSUB 620
      PRINT USING edit$; occurrence%
WEND
```

```
620   rem-----count occurrences---------
      location% = 1
      occurrence% = 0
      WHILE TRUE%
            location% = MATCH(arg$,object$,location%)
            IF location% = 0 THEN RETURN
            occurrence% = occurrence% + 1
            location% = location% + 1
      WEND
END
```

Examples:

```
MATCH("is","Now is the",1) returns 5

MATCH("##","October 8, 1976",1) returns 12

MATCH("a?","character",4) returns 5

MATCH("\#","123#45",1) returns 4

MATCH("ABCD","ABC",1) returns 0
```

The third example returns a 5 instead of a 3 because the starting position for the MATCH is position 4. In example four, the \ causes the # to MATCH only another #. Without the \ a 1 is returned.

The next example is a more complicated statement using the \:

```
MATCH("\#1\\\?","1#1\?2#",1)
```
returns 2

MID$ Function

MID$ returns a string consisting of the j% characters of a$ starting at the i% character.

Syntax:

b$ = MID$(a$, i%, j%)

Explanation:

If i% is greater than the length of a$, a null string is returned. If j% is greater than the length of a$, all characters from i% to the end of a$ are returned. If either i% or j% is negative, an error occurs. If i% is zero, a run-time error occurs. A zero value of j% returns a null string.

a$ must be a string expression; otherwise, an error occurs. i% and j% must be numeric. If i% or j% are real, they are converted to integers; if either i% or j% are strings, an error occurs.

Examples:

```
DIGIT$ = MID$(OBJECT$,POS%,1)

DAY$ = MID$("MONTUEWEDTHUFRISATSUN",DAY%*3-2,3)
```

NEXT Statement

A NEXT statement denotes the end of the closest unmatched FOR statement.

Syntax:

NEXT [*identifer* {,*identifer*}]

Explanation:

If the optional identifier is present, it must match the index variable of the terminated FOR statement.

The list of identifiers allows terminating multiple FOR statements. The statement number of a NEXT statement appears in an ON or GOTO statement, discussed later in this section, where execution of the FOR loop continues with the loop variables assuming their current values.

Examples:

```
FOR I% = 1 TO 10
      FOR J% = 1 TO 20
            X(I%,J%) = I% + J%
NEXT J%, I%

FOR LOOP% = 1 TO ARRAY,SIZE%
      GOSUB 200
      GOSUB 300
NEXT
```

ON Statement

The ON statement transfers execution to one of a number of labels.

Syntax:

ON *numeric exp* GOTO
stint number {, *stint number*}

ON *numeric exp* GOSUB
stint number {, *stint number*}

Explanation:

In an ON statement, the expression is used to select the statement number where execution continues. If the expression evaluates to 1, the first statement number is selected, and so forth. However, with an ON.. .GOSUB statement, the address of the statement following the ON statement is saved on the return stack. If the expression is less than one or greater than the number of statement numbers in the list, a runtime error occurs.

The expression must be numeric. A string expression generates an error. Integer expressions improve execution speed. If a real value is used, it is rounded to the nearest integer before selecting the statement number in which to branch.

Examples:

```
ON I% GOTO 10, 20, 30

ON J% - 1 GO SUB 12.10, 12,20, 12.30, 12.40

WHILE TRUE%
      GOSUB 100   REM ENTER PROCESS DESIRED
      GOSUB 110   REM TRANSLATE PROCESS TO NUMBER
      IF PROCESS.DESIRED% = 0 THEN RETURN
      IF PROCESS.DESIRED% < 6 THEN\
      ON PROCESS.DESIRED% GOSUB \
            1000, \ADD A RECORD
            1010, \ALTER NAME
            1020, \UPDATE QUANTITY
            1030, \DELETE A RECORD
            1040, \CHANGE COMPANY CODE
            1050, \REM GET PRINTOUT
      ELSE GOSUB 400 REM ERROR - RETRY
WEND
```

OPEN Statement

The OPEN statement activates an existing file for reading or updating.

Syntax:

OPEN "*filespec*" [RECL *rec length*]\
AS *file number* [BUFF *number of sectors*]\
RECS *size of sectors*

Explanation:

The first expression represents the filename on disk. The name can contain an optional drive reference. If the drive reference is not present, the currently logged drive is used. The filename must conform to the CP/M format for unambiguous filenames. Lower-case letters used in filenames are converted to upper-case. The expression must be string; if it is numeric, an error occurs. The following examples are valid filenames:

```
ACCOUNT.MST

CBAS86.CMD

B:INVENTORY.BAK
```

The third example shows a reference to a file on drive B.

The directory on the selected drive is searched and the named file is opened. If the file is not found in the directory, it is treated as if an end-of-file was encountered during a READ. When you specify a drive, it is your responsibility to ensure that the drive is available on your system.

When the optional RECL expression is present, the file consists of fixed length records. If the record length is negative or zero, a run-time error occurs. A file is accessed randomly or sequentially when a record length is specified; otherwise, only sequential access is allowed. The RECL expression must be numeric; real numbers are converted to integers. A string value causes an error.

The AS expression assigns an identification number to the file being opened. This value is used in future references to the file. Each active file must have a unique number assigned to it. If the expression is not between 1 and 20, a run-time error occurs. The expression must be numeric; real numbers are converted to integers. A string value causes an error.

The BUFF and RECS expressions are optional. If used, both must be present. The expression following BUFF specifies the number of disk sectors from the selected file to maintain in memory at one time.

If the expression is omitted, a value of one is assumed. The expression following RECS must be present when the BUFF expression is used, but the value of the expression is ignored. The value should be the size of a disk sector, usually 128 bytes.

If random access is used with a file, the BUFF expression must evaluate to 1; otherwise, a run-time error occurs.

Both expressions must be numeric; if it is a string value, an error occurs. Real numbers are converted to integers.

Twenty files can be active at once. Buffer space for files is allocated dynamically. Therefore storage space is saved by opening files as required and closing them when no longer needed.

Examples:

```
555 OPEN "TRANS.FIL" AS S

OPEN FILE.NAME$ AS FILE.NO% BUFF 26 RECS 128

OPEN WORK.FILE.NAME$(CURRENT.FILE%) \
      RECL WORK.LENGTH% AS CURRENT.FILE% BUFF BS% RECS 128
```

OUT Statement

The OUT statement sends the low-order eight bits of the second expression to the CPU output port selected by the low-order eight bits of the first expression.

Syntax:

OUT i%, j%

Explanation:

Both arguments must be numeric; they must be in the range of 0 to 255 for the results to be meaningful. If either expression is string, an error occurs. Real values are converted to integers before performing an OUT instruction.

Examples:

```
OUT 1,58

OUT FRONT.PANEL%, RESULT%

IF X% > 5 THEN OUT 8, ((X*X)-1,)/2.

OUT TAPE.DRIVE.CONTROL,PORT%, REWIND%

OUT PORT%(SELECTED%), ASC("$")
```

PEEK Function

PEEK returns the contents of the memory location specified by an absolute address.

Syntax:

i% = PEEK (*numeric exp.*)

Explanation:

The value returned is an integer ranging from 0 to 255. The memory location must be within the address space of your computer for your results to be meaningful.

The expression must be numeric. If a string expression is specified, an error occurs. Real values are rounded to the nearest integer.

Examples:

```
100 MEMORY%=PEEK(1)

FOR INDEX% = 1 TO PEEK%(BUFFER%)
      IN.BUFFER$(INDEX%) = CHR$(PEEK%(BUFFER%+INDEX%))
NEXT INDEX%
```

POKE Statement

POKE stores the low-order byte of a specified variable into memory at a specified absolute address.

Syntax:

POKE *numeric exp* , j%

Explanation:

The first expression must evaluate to an absolute address for your results to be meaningful.

Both expressions must be numeric. If a string expression is specified, an error occurs. Real values are rounded to the nearest integer.

Arguments are passed to machine language subroutines with the PEEK and POKE instructions.

Examples:

```
750 POKE 1700,ASC(”$”)

FOR LOC% = 1 TO LEN(OUT,MSG$)
      POKE MSG.LOC%+LOC%, ASC(MID$(OUT.MSG$,LOC%,1))
NEXT LOC%
```

POS Function

POS returns the next position to be printed on the console or the line printer. This value ranges from 1 to the line width currently in effect.

Syntax:

i= POS

Explanation:

If a LPRINTER statement is in effect, POS returns the next position to be printed on the printer. POS returns the actual number of characters sent to the output device. If cursor control characters are transmitted, they are counted even though the cursor is not advanced.

Examples:

```
PRINT “THE PRINT HEAD IS AT COLUMN:”; POS

IF (WIDTH.LINE - POS) < 15 THEN PRINT
```

PRINT Statement

The PRINT statement outputs the value of each expression to the console.

Syntax:

PRINT *expression delim*
{ *expression delim* }

Explanation:

If an LPRINTER statement is in effect, the output is directed to the list device. If the length of the numeric item results in the line width being exceeded, the number to be printed begins on the next line. Strings are output until the line width is reached and then the remainder of the string, if any, is output onto the next line.

The delimiter between expressions can be either a comma or a semicolon. The comma causes automatic spacing to the next column that is a multiple of 20. If this spacing results in a print position greater than the currently specified width, printing continues onto the next line. A semicolon outputs one blank after a number, and no spacing occurs after a string.

A carriage return and a line-feed are automatically printed when the end of a PRINT statement is encountered, unless the last expression is followed by a comma or a semicolon. These partial lines are not terminated until one of the following conditions occur:

- Another PRINT statement, whose list does not end in either a comma or semicolon, is executed.
- The line width is exceeded.
- A LPRINTER or CONSOLE statement is executed.
- The program executes a STOP statement.

A PRINT statement with no expression list causes a carriage return and a line-feed to be printed.

If execution of a program is ended due to an error, a carriage return and a line-feed are output.

Examples:

```
PRINT AMOUNT.PAID

PRINT QUANTITY, PRICE, QUANTITY * PRICE

PRINT “TODAY’S DATE IS: ”;MONTH$;“ ”;DAY%;“, ”;YEAR%
```

PRINT # Statement

The PRINT # statement outputs data to a disk file.

Syntax:

PRINT # <*file nunber*>[<*rec number*>];<*expression*>
{ ,<*expression*>}

Explanation:

The PRINT # statement writes expressions to the file specified by the file number. Each PRINT # statement executed creates a single record. Each expression used in the PRINT # statement creates a single field.

Use any number of expressions with the PRINT # statement and separate each one with a comma.

You can specify a random access record number for files that have a fixed record length. However, the amount of data written to fixed-length records must not exceed the record length specified in the RECL parameter in the CREATE or OPEN statement. You must add two bytes for the carriage return line-feed when determining the amount of data you can print to a record. Record numbers start with one, not zero.

Refer to Section 5 for more information on using disk files.

Examples:

```
CREATE “FILE.l” AS 1
      A$ = “FIELD.ONE”
      B% = 22222
PRINT #1; A$, B%

REM STORE CURRENT VALUE IN RECORD 5
OPEN “UPDATE.DAT” RECL 10 AS 15
      INPUT “Enter current value.”; VALUE%
PRINT #15,5; VALUE
```

PRINT USING Statement

The PRINT USING statement allows you to specify special formats for output data. The PRINT USING # variation directs formatted output to a disk file.

Syntax:

PRINT USING <*format string*>; *expression*, {, *expression*}

PRINT USING <*format string*>;#<*file number*> [, <*rec number*>];
<e*xpression*> {,<*expression*>}

Explanation:

The format string is a model for the output. A format string contains data fields and literal data. Data fields can be numeric or string types. Any character in the format string that is not part of a data field is a literal character. Format strings cannot be null strings. Table 3-1 describes characters that have special meaning in format strings.

Table 3-1 Special Characters in Format Strings

Character	Meaning
!	single-character string field
&	variable-length string field
/	fixed-length string field delimiter
#	digit position in numeric field
**	asterisk fill in numeric field
$$	float a $ in numeric field
.	decimal point position in numeric field
-	leading or trailing sign in numeric field
^	exponential position in numeric field
,	place comma every third digit before decimal point
\	escape character

The expression list tells which variables hold the data to be formatted. Separate each variable with a comma or semicolon. The comma does not cause automatic tabbing as it does with unformatted printing. PRINT USING matches each variable in the list with a data field in the format string. If there are more expressions than there are fields in the format string, execution is reset to the beginning of the format string.

While searching the format string for a data field, the type of the next expression in the list, either string or numeric, determines which data field to use. Section 5.3 has additional information on formatted printing.

Examples:

```
PRINT USING "###"; I%

ST$= "Total amount due is $$#,###.##"

PRINT USING ST$; TOTAL.DUE

PRINT USING "! ! !"; #15; "ALPHA", "BETA", "GAMMA"
```

RANDOMIZE Statement

The RANDOMIZE statement initializes or seeds the random number generator .

Syntax:

RANDOMIZE

Explanation:

The time taken for a user to respond to an INPUT statement is used to seed the random number generator. This amount of time can vary considerably with each execution of a program. Therefore, for RANDOMIZE to work correctly, it must be preceded by an INPUT statement .

Examples:

```
450 RANDOMIZE
```

```
RANDOMIZE
```

READ Statement

The READ statement assigns items listed in a DATA statement sequentially to the variables listed in the READ statement.

Syntax:

READ <*variable*> ,{<*variable*>}

Explanation:

READ statements can contain mixed DATA types as long as each type corresponds positionally to each type listed in the DATA statement. See DATA statement for more information.

Examples:

```
READ A%, B%, C%
DATA 14, 256, 73

READ A$, B%, C
DATA “Friday”, 25 , 4.28
```

READ # Statement

The READ # statement reads data fields from a specified disk file into variables.

Syntax:

READ # <*file number*>[,<*rec number*>];<*variable*>
{,<*variable*> }

Explanation:

The READ # statement reads expressions from a disk file specified by the file number. The file number is a unique identification number assigned to a file in the CREATE or OPEN statement. [File numbers are limited by the current implementation value for the number of files allowed open at one time.] Each READ # statement executed reads data sequentially, field by field, into the variables. READ # assigns one field of data to each variable. When reading a fixed file, the number of variables in the READ # statement must be less than or equal to the number of fields in each record.

You can specify a random access record number for files that have a fixed record length. Record numbers start with one, not zero.

Refer to Section 5 for more information on using disk files.

Examples:

```
OPEN "B:FILE.DAT" AS 8
WHILE NUMBER.OF.FIELDS%
      READ #8; FIELDS$
      PRINT FIELDS%
      NUMBER.OF.FIELDS% = NUMBER.OF.FIELDS% - 1
WEND

REM READ RECORD 3...FIELDS ONE AND TWO
IF END # 15 THEN 700
OPEN "FILE.1" AS 15
      READ #15, 3; FIELDl$, FIELD2$
```

READ # LINE Statement

The READ # LINE statement reads one complete line of data from a file and assigns the information to a string variable.

Syntax:

READ # <*file number*>, [<*record number*>];
LINE <*string variable*>

Explanation:

You can use only one variable after the keyword LINE. The variable must be a string variable.

The READ # LINE statement can read records accessed sequentially or randomly.

Examples:

```
READ #FILE.NO%; LINE D$

READ #F%, REC%; LINE X$
```

REM Statement

The REM statement documents a program.

Syntax:

REM [*string terminated with* CR]

REMARK [*string terminated with* CR]

Explanation:

REM statements do not affect the size of the program compiled or executed. When the Compiler ignores a REM statement, compilation continues with the statement following the next carriage return. A continuation character causes the next line to be part of the remark. An unlabeled REM statement can follow any statement on the same line. The statement number of a remark can be used in a GOSUB, GOTO, IF, or ON statement.

Examples:

```
REM THIS IS A REMARK

remark this is also a remark

TAX = 0.15 * INCOME     REM LOWEST TAX RATE REM \
      THIS SECTION CONTAINS THE \
      TAX TABLES FOR CALIFORNIA
```

The last example shows a REM statement on the same line with another statement. When using the REM statement in this manner, the colon is optional between the two statements. In all other cases involving multiple statements on the same line, the colon must separate the statements. If the REM statement is used on the same line with other statements, it must be the last statement on the line. All statements after a REM are ignored.

RENAME Function

The RENAME function changes the name of the file specified by b$ to the name given by a$. Renaming a file to a name that already exists produces a rum-time error.

Syntax:

i% = RENAME(a$, b$)

Explanation:

The RENAME function returns an integer value. A true (-1) is returned when the RENAME is successful, and a false (0) is returned when the RENAME fails. For example, false is returned if b$ does not exist.

A file must be closed before it is renamed; otherwise, when CBASIC automatically closes files at the end of processing, it attempts to close the renamed file under the name with which it was opened. This causes a run-time error because the original filename no longer exists in the CP/M file directory.

Both arguments must be string. If either a$ or b$ is numeric, an error occurs.

The RENAME function allows you to use the following back-up convention:

1. The output file is opened with a filetype of $$$, indicating it is temporary.
2. Any file with the same filename as the output file, but with a filetype BAK, is deleted.
3. Data is written to the temporary file as the program is processed.
4. At the end of processing, the program renames any file with identical filename and filetype as the output file to the same filename, but with the filetype BAK.
5. The program renames the temporary output file to the proper filename and filetype.

Examples:

```
DUMMY% = RENAME("PAYROLL.MST","PAYROLL.$$$")

IF RENAME(NEWFILE$,OLDFILE$) THEN RETURN
```

RESTORE Statement

A RESTORE statement allows rereading of the constants contained in DATA statements.

Syntax:

RESTORE

Explanation:

A RESTORE statement repositions the DATA statement pointer to the beginning of the DATA area. A RESTORE statement is executed when a CHAIN statement is present.

Examples:

```
500 RESTORE

IF END.OF.DATA% THEN RESTORE
```

RETURN Statement

The RETURN statement sends control from a subroutine back to the main program.

Syntax:

RETURN

Explanation:

The RETURN statement causes the execution of a program to continue at the location on top of the return stack. The call might be a GOSUB statement , an ON.. . GOSUB statement, or a multiple-line function call. (See Section 4.2.2 for a discussion of multiple-line functions.)

If a return is executed without previously executing a GOSUB, ON.. .GOSUB, or multiple-line function call, a run-time error occurs.

Examples:

```
500 RETURN

IF ANSWER.VALID% THEN RETURN
```

RIGHT$ Function

RIGHT$ returns a string consisting of the i% rightmost characters of a$.

Syntax:

b$ = RIGHT$(a$, i%)

Explanation:

If i% is negative, a run-time error occurs. If i% is greater than the length of a$, the entire string is returned. If i% is zero, a null string is returned.

a$ must evaluate to a string; otherwise, an error occurs. i% must be numeric. If i% is real, it is converted to an integer. If i% is a string, an error occurs.

Examples:

```
IF RIGHT$(ACCOUNT.NO$,1) = "0" THEN \
      TITLE.ACCT% = TRUE%

NAME$ = RIGHT$(NAME$,LEN(NAME$)-LEN(FIRST.NAME$))
```

RND Function

RND generates a uniformly distributed random number between 0 and 1. The value returned by RND is a real number.

Syntax:

x= RND

Explanation:

To avoid identical sequences of random numbers each time a program is executed, the RANDOMIZE statement is used to seed the random number generator.

Examples:

```
DIE%=INT%(RND*6.)+1

IF RND > .5 THEN \
      HEADS% = TRUE% \
ELSE \
      TAILS% = TRUE%
```

SADD Function

The SADD function returns the address of a specified string.

Syntax:

i% = SADD (<*string variable*>)

Explanation:

Strings are stored as a sequential list of ASCII characters. The first two bytes hold the length of the string followed by the actual ASCII values. The length is stored as an unsigned binary integer. SADD returns an integer equal to the address of the first byte of the length.

If the expression is a null string, SADD returns a zero.

Examples:

The following statements place the address of STRING$ into the address stored in PARM.LOC%:

```
POKE PARM.LOC%,SADD(STRING$) AND 0FFH

POKE PARM.LOC%+l,SADD(STRING$)/256
```

SAVEMEM Statement

The SAVEMEM statement reserves space for a machine language subroutine and loads the specified file during execution. Only one SAVEMEM statement can appear in a program.

Syntax:

SAVEMEM <*constant*> , <*filespec*>

Explanation:

The constant must be an unsigned integer specifying the number of bytes of space to reserve for machine language subroutines. The space is reserved in the highest address space of the CP/M Transient Program Area. The beginning address of the reserved area is calculated by taking the constant specified in the SAVEMEM statement and subtracting it from the 16-bit address stored by CP/M at absolute address 6 and 7. However, for the 8086 version of CBASIC, the load address is rounded down to the nearest 16-byte paragraph boundary.

The expression must be string, and can specify any valid unambiguous filename. The selected file is loaded into memory starting with the address calculated above. Records are read from the file until either an end-of-file is encountered, or the next record to be read overwrites a location above the Transient Program Area.

If the constant specifies less than 128 bytes to be saved, nothing is read, but the space is still reserved. If the expression is a null string, space is saved but no file is loaded.

If a main program has a SAVEMEM statement, any chained program having a SAVEMEM statement must reserve the same amount of space. Each chained program loads a new machine language file or uses the file loaded by a previous program. The space reserved by the main program is not reclaimed by a subsequent program.

It is your responsibility to ensure the machine language routines are assembled to execute at the proper address. Also, the location where a program is loaded depends on the size of the CP/M system used.

The CALL statement accesses routines loaded by SAVEMEM. The CALL statement loads the data, code, and extra segment registers to the base of the SAVEMEM area.

Examples:

```
SAVEMEM 256, “SEARCH.CMD”

SAVEMEM 512, DR$+ “CHECK.” + ASSY$(FN.CPM.SIZE%)
```

SGN Function

SGN (x) returns an integer value representing the algebraic sign of the argument.

Syntax:

i% = SGN(*numeric exp*)

Explanation:

SGN returns a -1 if x is negative, a 0 if x is zero, and a ± 1 if x is greater than zero.

x can be integer or real. Integer values of x are converted to real numbers. The argument should be numeric. SGN always returns an integer.

Examples:

```
IF SGN(BALANCE) <> 0 THEN \
      OUTSTANDINGBAL% = TRUE%

IF SGN(BALANCE) = -1 THEN \
      OVERDRAWN% = TRUE%
```

SIN Function

SIN(x) returns the sine of x.

Syntax:

y = SIN(x)

Explanation:

The argument x is expressed in radians. The value returned by SIN is real. If x is an integer, it is converted to a real number.

Examples:

```
FACTOR(Z) = SIN(A - B/C)

IF SIN(ANGLE/(2.0 # PI)) = 0.0 THEN \
      PRINT "HORIZONTAL"
```

SIZE Function

SIZE returns the size in 1 kilobyte blocks of the file specified by a$.

Syntax:

i% = SIZE(a$)

Explanation:

If the file is empty or does not exist, zero is returned. a$ is any CP/M ambiguous filename.

The argument must be a string expression. If the argument is numeric, an error occurs. The SIZE function returns an integer.

The SIZE function returns the number of blocks of disk space used by the file or files referred to by the argument. When the operating system allocates disk space to a file, it does so in one block increments. A file of 1 character occupies a full block of space. This means the SIZE function returns the amount of space reserved by the file rather than the size of the data in the file.

This function is useful in a program that duplicates or constructs a file on disk. If the program creates a file of a given size, dependent on the size of its input file, it first determines whether or not there is sufficient free space on the disk before building the new file.

Consider a program that reads a file named INPUT from drive A, processes the data, and then writes a file named OUTPUT to drive B. Assume the size of OUTPUT is 125% of INPUT. The following routine ensures space is available on disk B before processing:

```
rem------test for enough room------
size.of.output% = 1.25 * size("A:INPUT")
free,blocks% = 241 - size("B:*.*")
if free.space% < size.of.output% then\
      enough.room% = FALSE%\
      else enough.room% = TRUE%
return
```

Examples:

```
SIZE("NAMES.BAK")

SIZE(COMPANY$ + DEPT$ + ".NEW")

SIZE("B:ST?RTR?K.*")

SIZE("*.*")

SIZE("*.BAS")
```

SQR Function

SQR (x) returns the square root of the argument x.

Syntax:

y = SQR(x)

Explanation:

If x is negative, a warning message is printed and the square root of the absolute value of the argument is returned.

The value returned by SQR is real. If x is an integer, it is converted to a real number.

Examples:

```
HYPOT = SQR((SIDE1^2.0)+(SIDE2^2.0))

PRINT USING \
      “THE SQR ROOT OF X IS: ####.##”; SQR(X)
```

STOP Statement

A STOP statement terminates program execution.

Syntax:

STOP

Explanation:

All open files are closed, the print buffer is emptied, and control returns to the host system. Any number of STOP statements can appear in a program.

Examples:

```
400 STOP

IF STOP.REQUESTED THEN STOP
```

STR$ Function

STR$ (x) returns the character string representing a numeric value specified by x.

Syntax:

a$ = STR$(x)

Explanation:

If x is a string, an error occurs.

Examples:

```
PRINT STR$(NUMBER)

IF LEN(STR$(VALUE))>5 THEN ED$="#######"
```

TAB Function

TAB causes the cursor or list device print head to be positioned at a location specified by the value of the expression. The TAB function is only used in PRINT statements.

Syntax:

TAB (*numeric exp*)

Explanation:

If the value of the expression is less than or equal to the current print position, a carriage return and line-feed are output then TAB is executed.

The TAB predefined function is implemented by outputting blank characters until the desired position is reached. If cursor or printer control characters are output, the cursor or print head might be positioned incorrectly.

The expression must be numeric. If a string expression is specified, an error occurs. If the expression is real, it is first rounded to an integer. An error occurs if the expression is greater than the current line width.

Examples:

```
PRINT TAB(15);“X”

PRINT “THIS IS COL. 1”;TAB(50);“THIS IS COL. 50”

PRINT TAB(X%+Y%/Z%);“!”;TAB(POS%+OFFSET%);

PRINT TAB(LEN(STR$(NUMBER)));“*”
```

TAN Function

TAN (x) returns the tangent of the argument x.

Syntax:

y = TAN(x)

Explanation:

The argument x is expressed in radians. The value returned by TAN is real. If x is an integer, it is converted to a real number.

Examples:

```
POWER.FACTOR = TAN(PHASE.ANGLE)

QUIRK = TAN(X - 3.0 * COS(Y))
```

UCASE$ Function

UCASE$ translates lower-case characters to upper-case.

Syntax:

b$ = UCASE$(a$)

Explanation:

UCASE$ returns a string where the lower-case characters in the argument a$ are translated to upper-case; other characters are not altered. A$ remains unchanged unless a$ is set equal to UCASE$(a$).

The value returned by UCASE$ is a string. If a$ is numeric, an error occurs.

Examples:

```
IF UCASE$(ANS$) = “YES” THEN\
      RETURN \
ELSE STOP

NAME$ = UCASE$(NAME$)
```

VAL Function

VAL converts a numeric character string into a real number.

Syntax:

x= VAL(a$)

Explanation:

VAL converts the argument a$ into a floating-point number. Conversion continues until a character is encountered that is not part of a valid number, or until the end of the string is encountered.

If a$ is a null string, or the first nonblank character of a$ is not a +, -, or digit, a zero is returned.

The argument must be a string; otherwise, an error occurs.

Examples:

```
PRINT ARRAY$(VAL(IN.STRING$))

ON VAL(PROG.SEL$) GOSUB 10, 20, 30, 40, 50
```

VARPTR Function

VARPTR returns the storage location assigned to the variable by the run-time monitor.

Syntax:

i% = VARPTR (*variable*)

Explanation:

With an unsubscripted numeric quantity, this is the actual location of the variable in question. For string variables, the value returned by VARPTR is the address of a 16-bit pointer to the referenced string. Because strings are dynamically allocated the actual location of the string varies, but the value returned by VARPTR remains unchanged during program execution. If the variable is in common, the location returned by VARPTR remains unchanged after chaining.

If the variable is subscripted, the value returned by VARPTR is the address of a pointer to the array dope vector in the free storage area. The array follows the dope vector. The first byte of the dope vector is the number of dimensions followed by n1, n is the number of dimensions, 16-bit offsets into the array. The final 16-bit quantity in the dope vector is the number of entries in the array. The array follows in row-major order.

WEND Statement

A WEND statement denotes the end of the closest unmatched WHILE statement.

Syntax:

WEND

Explanation:

A WEND statement must be present for each WHILE statement in a program.

Branching to a WEND statement is the same as branching to its corresponding WHILE statement.

Examples:

```
WHILE -1
      PRINT "X"
WEND

WHILE X > Z
      PRINT X
      X= X - 1.0
WEND

TIME = 0.0
TIME.EXPIRED% = FALSE%
WHILE TIME < LIMIT
      TIME = TIME + 1.0
      IF CONSTAT% THEN \
            RETURN REM ANSWERED IN TIME
WEND
TIME.EXPIRED% = TRUE%
RETURN

WHILE ACCOUNT.IS.ACTIVE%
      GOSUB 100   REM ACCUMULATE INTEREST
WEND
```

```
WHILE FILE.EXISTS%
      WHILE TRUE%
            IF ARG$ = ACCT$ THEN \
                  ACTIVITY% = TRUE% :\
                  RETURN
            IF ARG$ < ACCT$ THEN \
                  ACTIVITY% = FALSE% :\
                  RETURN
            GOSUB 3000  REM READ ACCT$ REC
      WEND
WEND
ACTIVITY% = FALSE%
RETURN

WHILE TRUE%
      INPUT LINE STRING$
      IF STRING$ = CONTINUE$ THEN RETURN
WEND
```

WHILE Statement

The WHILE statement loops program control until the specified expression evaluates to zero.

Syntax:

WHILE *expression*

Explanation:

Execution of all statements between the WHILE statement and its corresponding WEND are repeated until the value of the expression, in the WHILE section, is zero. If the value is zero initially, the statements between the WHILE and WEND are not executed. Variables used in the WHILE expression can change during execution of the loop.

The expression should be integer. This reduces execution time and also reduces the intermediate code generated by the Compiler. If the expression is real, the value is rounded and converted to an integer. A string expression results in an error.

End of Section 3

Section 4
Defining and Using Functions

Functions are useful when the same routine or computation is needed in a number of locations in a program. Once defined, a function can be referenced or called any number of times in the program.

All CBASIC functions return a value. Any routine that results in a value, either string or numeric, can be defined as a function. Functions can pass values and parameters for use at each invocation.

4.1 Function Names

Function names are defined with the letters FN followed by any combination of numbers, letters, or periods. Any number of characters can be used in a function name; however, only the first 31 characters, including the FN, distinguish one name from another. A function name cannot contain spaces.

The type of function name determines the type of value that the function passes back to the main program.

- Names for string functions end with $.
- Names for integer functions end with %.
- Names for real number functions do not end with $ or %.

You must use a function name to define a function and to reference a function from another location in a program. The following examples are valid function names:

```
FN.PROPER.FUNCTION.NAMES

FN.TRUNCATE$

FN51234%
```

4.2 Function Definitions

A function definition must occur in a program before making a function reference. To define a function, the keyword DEF must precede the function name. CBASIC supports two types of function definitions: single-statement and multiple-statement.

4.2.1 Single-Statement Functions

Single-statement function definitions use an equal sign followed by an expression. The expression contains the actual process that the single-statement function is coded to perform. The data types used in the expression must correspond to the data type used in the function name. Use the following format when defining single-statement functions:

DEF FN*function.name* [(dummy arg list)] = expression

A dummy argument holds a place for a variable that is specified in a function reference. A dummy argument is either a string variable or a numeric variable; it is never a constant. The dummy argument must have the same data type as the variable used in the function reference. However, the data type for the dummy argument is independent of the function name type. CBASIC considers dummy arguments local to the function. Local variables are independent of the rest of a program. CBASIC parameters are passed by value.

The following examples show single-statement function definitions:

```
DEF FN25 = RND * 25.0

100 DEF FN.HYPOT(SIDE1,SIDE2)= \
      SQR((SIDE1 * SIDE1) + (SIDE2 * SIDE2))

DEF FN.LEFT.JUSTIFY$(A$,LEN%)=LEFT$(A$+BLNKS$,LEN%)
```

4.2.2 Multiple-Statement Functions

Multiple-statement function definitions use a series of CBASIC statements preceded by a DEF statement and terminated with a FEND statement. Also, a RETURN statement is placed at some point in the body of the function. The RETURN statement ends function execution and sends control back to the main program. Use any number of RETURN statements, but be sure a FEND statement is the last statement that appears in a multiple-statement function. Use the following format when defining multiple-statement functions:

```
DEF FN function.name [(dummy arg list)]
    CBASIC Statements
    .
    .
    .
    RETURN
FEND
```

The following two examples show multiple-statement function definitions:

```
DEF FN.READ.INPUT(INPUT.NO%)
      READ # INPUT.NO%; CUSTNO%, AMOUNT
      RETURN
FEND
200 DEF FN.COUNT%(INDEX1%)
      COUNT% = 0
      FOR I% = 1 TO INDEX1%
            COUNT% = COUNT% + ARRAY(I%)
      NEXT I%
      FN.COUNT% = COUNT%
      RETURN
FEND
```

The following rules apply to multiple-statement functions:

- DEF and COMMON statements cannot appear in a function definition.
- GOTO statements that reference a line outside of the function are not allowed.
- The DIM statement allocates a new array upon each execution of a function. Data stored in an array from a previous execution is lost. Arrays in multiple-statement functions are global to an entire program.
- Functions cannot be nested. However, a function can be called from within another function.

4.3 Function References

User-defined functions can be referenced in any CBASIC statement or expression. Be sure to specify the same number of parameters in the function reference that are specified in the function definition. The function substitutes the current value of each expression in the reference statement for the dummy argument in the function definition. The following are examples of function references:

```
300 PRINT FN.A(FN.B(X))

IF FN.LEN%(“INPUT DATA”,X$,Q) < LIMIT% THEN
      GOSUB 3000

WHILE FN.ALTITUDE(CURR.ALT%) > MINIMUM.SAFE
      CURR.ALT%=INP(ALT.PORT%)
WEND
```

End of Section 4

Section 5
Input and Output

CBASIC uses the operating system to control input and output for interaction between programs, consoles or terminals, printers, and disk drives.

5.1 Console Input and Output

CBASIC performs all console input and output directly. You can use the following statements and functions to input data from a console device. Refer to Section 3 for more detailed descriptions of each statement and function.

- INPUT statements query the user for information during program execution. You can enter any number of input values with an INPUT statement. You can have a prompt message displayed if you desire.
- INPUT LINE works like an INPUT statement, but accepts only one variable for data to be entered. All characters entered in response to INPUT LINE are interpreted as one string.
- CONSTAT% is a predefined function that determines console status. The function returns a logical true value (-1) if a character is ready at the console, and a logical false value (0) if a character is not ready.
- CONCHAR% is a function that waits for an entry from the keyboard and returns an eight-bit ASCII representation of the character entered. CONCHAR% echoes characters of ASCII decimal value greater than 31.

The following CBASIC statements and functions control console output .

- The CONSOLE statement restores printed output to the console device.
- The TAB predefined function moves the console cursor to a specified position on the screen. TAB also works with printers. Use TAB only in the PRINT statement.
- The POS predefined function returns the next available position on the console screen to be printed.

5.2 Printing

CBASIC provides three statements to control output to a line printer. Refer to Section 3 for additional information on each statement .

- PRINT outputs data to a console or printer.
- LPRINTER directs all output PRINT statements to the line printer or list device.
- PRINT USING allows formatting of printed output to the console or printer.

5.3 Formatted Printing

The PRINT USING statement allows you to specify special formats for output data. You can output formatted data to the console or line printer with CONSOLE or LPRINTER. The PRINT USING # variation directs formatted output to a disk file. Write a PRINT USING statement as follows:

```
PRINT USING <format string>;[#<file number>[,<rec number>];]
            <expression list>
```

The format string is a model or image for the output. A format string contains data fields and literal data. Data fields can be numeric or string type. Any character in the format string that is not part of a data field is a literal character. Format strings cannot be null string expressions. Table 5-1 describes characters hat have special meanings in format strings:

Table 5-1 Special Characters in Format Strings

Character	Meaning
!	single-character string field
&	variable-length string field
/	fixed-length string field delimiter
#	digit position in numeric field
**	asterisk fill in numeric field
$$	float a $ in numeric field
.	decimal point position in numeric field
-	leading or trailing sign in numeric field exponential position in numeric field
,	place comma every third digit before decimal point
\	escape character

The expression list tells which variables hold the data to format. Separate each variable with a comma or semicolon. The comma does not cause automatic tabbing as it does with unformatted printing. PRINT USING matches each variable in the list with a data field in the format string. If there are more expressions than there are fields in the format string, execution resets to the beginning of the format string.

While searching the format string for a data field, the type of the next expression in the list, either string or numeric, determines which data field to use. For example, if PRINT USING encounters a numeric data field while outputting a string, the statement treats characters in the numeric data field as literal data. An error occurs if there is no data field in the format string of the type required.

5.3.1 String Character Fields

Specify a one-character string data field with an exclamation point, !. PRINT USING outputs the first character of the next expression statement list. For exanple,

```
F.NAME$="Lynn":M.NAME$ = "Marion":L.NAME$= "Kobi"
PRINT USING "!. !. &"; F.NAME$,M.NAME$,L.NAME$
```

outputs

```
L. M. Kobi
```

In this example, PRINT USING treats the period as literal data.

5.3.2 Fixed-length String Fields

Specify a fixed-length string field of more than one position with a string of characters enclosed between a pair of slashes. The width of the field is equal to the number of characters between the slashes, plus two. Place any characters between the slashes to serve as fill. PRINT USING ignores fill characters for fixed-length string fields.

A string expression from the print list is left-justified in the fixed field and, if necessary, padded on the right with blanks. PRINT USING truncates a string longer than the data field on the right. For example,

```
FOR1$ = "THE PART REQUIRED IS /...5....0....5/"
PART.DESCRP$ = "GLOBE VALVE, ANGLE"
PRINT USING FOR1$; PART.DESCRP$
```

outputs

```
THE PART REQUIRED IS GLOBE VALVE, ANG
```

Using periods and numbers between the slashes makes it easy to verify that the data field is 16 characters long. Periods and numbers do not effect the output.

5.3.3 Variable-length String Fields

Specify a variable-length string field with an ampersand, &. This results in a string output exactly as defined. For example,

```
COMPANY$ = “SMITH INC.”

PRINT USING “& &”; “THIS REPORT IS FOR”,COMPANY$
```

outputs

```
THIS REPORT IS FOR SMITH INC.
```

The following example shows how a string can be right-justified in a fixed-length string field using a variable-length string field.

```
FLD.S% = 20
BLK$ = “                    ”
PHONE$ = “408-649-3896”
PRINT USING “#&”; RIGHT$(BLK$ + PHONE$, FLD.S%)
```

outputs

```
#         408-649-3896
```

The preceding example uses the # as a literal character because the print list contains only a string expression. A # can also indicate a numeric data field.

5.3.4 Numeric Data Fields

Specify a numeric data field with a pound sign, #, to indicate each digit required in the resulting number. One decimal point can also be included in the field. Values are rounded to fit the data field. Leading zeros are replaced with blanks. When the number is negative, PRINT USING prints a minus sign to the left of the most significant digit. A single zero prints to the left of the decimal point for numbers less than one if you provide a position in the data field. For example,

```
X = 123.7546
Y = -21.0
FOR$ = “####.####   ###.#   ###”
PRINT USING FOR$; X, X, X
PRINT USING FOR$; Y, Y, Y
```

outputs

```
123.7546   123.8   124
-21.0000   -21.0   -21
```

Tell PRINT USING to print numbers in exponential format by appending one to four up arrows, ^, to the end of the numeric data field. For example,

```
X = 12.345
PRINT USING "#.###^^    "; X, -X
```

outputs

```
l.235E 01    -.123E 02
```

PRINT USING reserves four positions for the exponent regardless of the number of up arrows used in the field.

If one or more commas appear embedded in a numeric data field, the number prints with commas between each group of three digits that precede the decimal point. For example,

```
PRINT USING "##,###   "; 100, 1000, 10000
```

outputs

```
100   1,000   10,000
```

PRINT USING includes each comma that appears in the data field in the width of the field. You need only one comma to obtain embedded commas in the output; however, placing each comma in the data field at the specified position clarifies the formatting statement.

For example, the following data fields produce the same results, except the width of the first field allows only nine output digits. The second field allows ten digits.

```
#,########
#,###,###,###
```

Commas do not print if you use the exponent option. In this case, PRINT USING treats commas as pound signs, #.

You can use asterisk filling in a numeric data field by appending two asterisks to the beginning of the data field. You can float a dollar sign by appending two dollar signs to the beginning of the data field. Do not use the exponential format with either asterisk filling or floating dollar signs. PRINT USING includes a pair of asterisks or dollar signs in the count of digit positions available for the field. The asterisks or dollar signs appear in the output if there is enough space. The dollar sign does not print if the number is negative. For example,

```
COST = 8742937.56
PRINT USING "**##,######.##   "; COST, -COST
PRINT USING "$$##,######.##   "; COST, -COST
```

outputs

```
**8,742,937.56   *-8,742,937.56
 $8,742,937.56    -8,742,937.56
```

PRINT USING outputs a number with a trailing sign instead of a leading sign if the last character in the data field is a minus sign. A blank replaces the minus sign in the output if the number is positive. For example,

```
PRINT USING"###- ###^^^^- "; 10, 10, -10, -10
```

outputs

```
10  100E-0l    10- l00E-01-
```

PRINT USING fixes the sign position as the next output position if a minus sign is the first character in a numeric data field. If the number is positive, a blank prints instead of the minus sign. For example,

```
PRINT USING "-####   "; 10, -10
```

outputs

```
10   -   10
```

If a number does not fit in a numeric data field without truncating digits before the decimal point, a percent sign, %. precedes the number in the standard format. For example,

```
X = 132.71
PRINT USING "##.#   ###.#"; X,X
```

outputs

```
% 132.71  132.7
```

5.3.5 Escape Character

You can use a special format string character as literal data in a data field with the escape character. A backslash, \, signals PRINT USING to treat the next consecutive character as a literal character. For example, a pound sign, #, can precede a number. For example, ITEM.NUMBER = 31

```
PRINT USING "THE ITEM NUMBER IS\ ###"; ITEM.NUMBER
```

outputs

```
THE ITEM NUMBER IS #31
```

Two consecutive backslashes cause a one backslash to print as a literal character. An escape character cannot be the last character in a format string.

5.4 File Organization

CBASIC organizes information on a disk surface into three levels: files, records, and fields.

- FILES consist of one or more records.
- RECORDS are groups of fields. Each record is delimited by a carriage return and line feed.
- FIELDS are the individual data items. Each field within a record is delimited by a comma.

CBASIC supports two types of data files on disk: sequential and random access.

5.4.1 Sequential Files

Sequential or stream organization is performed on a strict field-by-field basis. The PRINT # statement writes each field to the disk in a continuous stream. Each data item uses only as much space as needed. Each PRINT # statement executed creates a single record. Each variable used in the PRINT # statement creates a single field. Individual record lengths vary according to the amount of space occupied by the fields. There is no padding of data space. The following diagram shows a sequential file composed of three records.

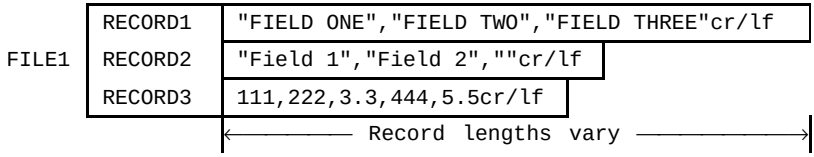

Figure 5-1 Sequential File

Field three in record two is a null string. Commas serve as delimiters, but are considered string characters when embedded in a pair of quotation marks. Quotation marks are also considered string characters when embedded in a pair of quotation marks.

The following CBASIC program creates the sequential file diagrammed above.

```
CREATE “FILE.1” AS 1
      A$ = “FIELD ONE”
      B$ = “FIELD TWO”
      C$ = “FIELD THREE”
      D$ = “FIELD 1”
      E$ = “FIELD 2”
      F$ = “”
      G% = 111
      H% = 222
      I  = 3.3
      J% = 444
      K  = 5.5
PRINT#1; A$, B$, C$
PRINT#1; D$, E$, F$
PRINT#1; G%, H%, I, J%, K
CLOSE1
END
```

The three PRINT statements correspond to the three records and each variable corresponds to a field.

When sequential files are accessed, each field is read consecutively one at a time from the first to the last. The READ# statement considers a field complete when it encounters a comma or a carriage-return and line feed. The following program reads the fields in FILE.1 sequentially and prints them on the console screen.

```
IF END #19 THEN 100
OPEN "FILE.1" AS 19
      FOR I% = 1 TO 11
            READ #19; FIELDS$
      PRINT FIELDS$
      NEXT I%
100 END
```

Any type of variable can be used in the READ# statement in a sequential access. Executing the preceding program outputs the following display on the screen.

```
FIELD ONE
FIELD TWO
FIELD THREE
FIELD 1
FIELD 2

111
222
3.3
444
5.5
```

5.4.2 Relative Files

Relative files offer the advantage of random access, which is the ability to access any record in a file directly. Record lengths are fixed. Data space between the end of the last field and the carriage-return line-feed is padded with blanks. The carriage-return and line-feed occupy the last two bytes of the record. The number of bytes occupied by the fields, field delimiters, and the carriage-return line-feed cannot exceed the specified record length. The maximum length of a record is 65,535 bytes. The following diagram shows a relative file composed of three records.

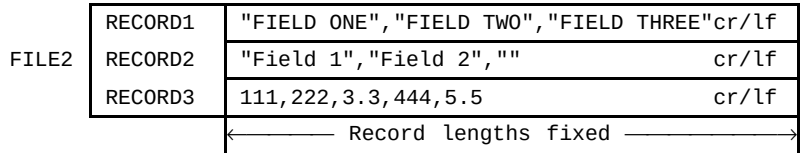

Figure 5-2 Relative File

The same rules regarding commas, quotation marks, and null strings in sequential files apply to relative files. The following program creates the relative file diagrammed above.

```
CREATE FILE.2 RECL 40 AS 2
A$ = “FIELD ONE”
B$ = “FIELD TWO”
C$ = “FIELD THREE”
D$ = “FIELD 1”
E$ = “FIELD 2”
F$ = “”
G% = 111
H% = 222
I  = 3.3
J% = 444
K  = 5.5
PRINT #2,1; A$, B$, C$
PRINT #2,2; D$, E$, F$
PRINT #2,3; G%, H%, I, J%, K
CLOSE 2
END
```

Random access to a relative file is accomplished by specifying a relative record number. A relative file can have a maximum of 65,535 records. Therefore, the maximum record number you can specify is 65535.

The relative record number is entered in all PRINT# and READ# statements after the file identification number. The two numbers are separated with a comma. In the following example, 5 is the relative record number.

```
PRINT #2,5; VARIABLE1%, VARIABLE2%
```

CBASIC locates each record on a randomly accessed file by taking the relative record number, subtracting 1, and multiplying that difference by the record length. The result is a byte displacement value for the desired record measured from the beginning of the file. The record to be accessed must be specified in each READ# or PRINT# statement executed. Each READ# and PRINT# statement executed accesses the next specified record. The following program reads the first three fields from record three in FILE.2.

```
IF END #20 THEN 200
OPEN "FILE.2" RECL 40 AS 20
      READ #20,3; FIELD1$, FIELD2$, FIELD3
      PRINT FIELD1$, FIELD2$, FIELD3
200 END
```

The data types of the variables in the READ# statement must match the data contained in the fields being read. Executing the above program outputs the following display on screen.

```
111               222          3.3
```

5.5 Maintaining Files

CBASIC uses the operating system file accessing routines to store and retrieve data in disk files. All data is represented in character format using the ASCII code. Programs can create, open, read, write, and close data files with the following CBASIC statements. Each statement is described in more detail in Section 3.

- CREATE originates a new data file on disk. The CREATE statement erases a preexisting file of the sane name before creating the new file.
- OPEN accesses an existing file for reading or updating. If the file does not exist, the program processes an end-of-file condition.
- FILE accesses an existing file for reading or updating. If the file does not exist, the FILE statement creates it.
- READ# accesses a specified file and assigns the data sequentially, field by field, into specified variables. Data can also be accessed from a specified record.
- PRINT# outputs data to a specified file and assigns the data sequentially into fields from specified variables. Data can also be accessed from a specified record.
- PRINT USING # outputs data to a specified file using formatted printing options.
- CLOSE deactivates a file from processing. The specified file is no longer available for input or output until reopened.
- DELETE deactivates a file from processing and erases it from the disk surface.

End of Section 5

Section 6
Machine Language Interface

CBASIC's machine level environment is somewhat advanced. To understand this section, you should have a working knowledge of CP/M, assembly language, and a familiarity with elementary computer architecture. Differences between CBASIC's 8-bit and 16-bit formats are most visible at the machine level.

6.1 Memory Allocation

The operating system loads the CBASIC run-time interpreter into the Transient Program Area (TPA) to execute CBASIC programs. The memory available in the TPA is partitioned into six areas of varying size. The following diagram shows memory allocation in the CP/M TPA. For the 8-bit (8080) version, addresses are absolute. In the 16-bit (8086) version, addresses are offsets from either the code or data segments.

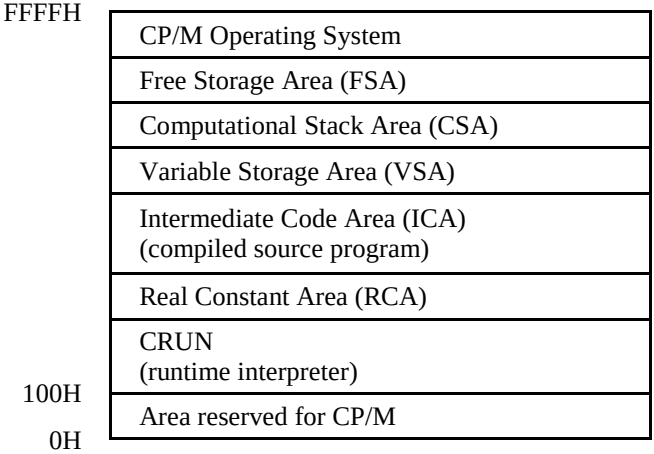

Figure 6-1 CP/M Memory Allocation

- The area extending from the base of memory up to hexadecimal address 100H is reserved for CP/M.
- CP/M loads the run-time Interpreter at the base of the TPA starting at 100H. The nest of the TPA is partitioned into five areas used by the run-time Interpreter during program execution.
- The Real Constant Area (RCA) holds all real numbers defined as constants in a CBASIC program. If a constant is used more than once in a program, it appears only once in the RCA. Real constants require eight bytes of storage space.
- The Intermediate Code Area (ICA) stores the intermediate code generated by the Compiler. The Interpreter fetches the actual computer instructions from the ICA during program execution.
- The Variable Storage Area (VSA) reserves space to store the current value of each variable in the program. The VSA contains all variables passed through COMMON statements to chained programs. COMMON variables always appear first in the VCA. The VCA reserves eight bytes of storage space for each variable, regardless of the data type. For array and string variables, the actual value of the variable is stored in the Free Storage Area. The value stored in the VSA points to the actual value in the FSA.
- The Computational Stack Area (CSA) is fixed at 160 bytes of memory. The CSA evaluates expressions and passes parameters to CBASIC predefined and user-defined functions. There is room to place 20 eight-byte real numbers on the stack.
- The Free Storage Area (FSA) stores arrays, strings and file buffers. Variably sized blocks of memory are allocated from the FSA as required and returned when no longer needed.
- The CP/M operating system tracks occupy the very top of memory. A 16-bit address at 0006H and 0007H points to the beginning of CP/M.

The starting and ending address for each partition in the TPA varies for different programs. Once allocated however, the amount of memory occupied by each partition remains fixed during program execution.

6.2 Internal Data Representation

CBASIC machine level representation varies somewhat for real numbers, integers, strings, and arrays.

- REAL NUMBERS are stored in binary coded decimal (BCD) floating-point form. Each real number occupies eight bytes of memory, as shown in the following diagram. The high-order bit in the first byte (byte 0) contains the sign of the number. The remaining seven bits in byte 0 contain a decimal exponent. Bytes 1 through 7 contain the mantissa. Two (BCD) digits occupy each of the seven bytes in the mantissa. The number's most significant digit is stored in high-order four bits of byte 7. The floating decimal point is always situated to the left of the most significant digit.

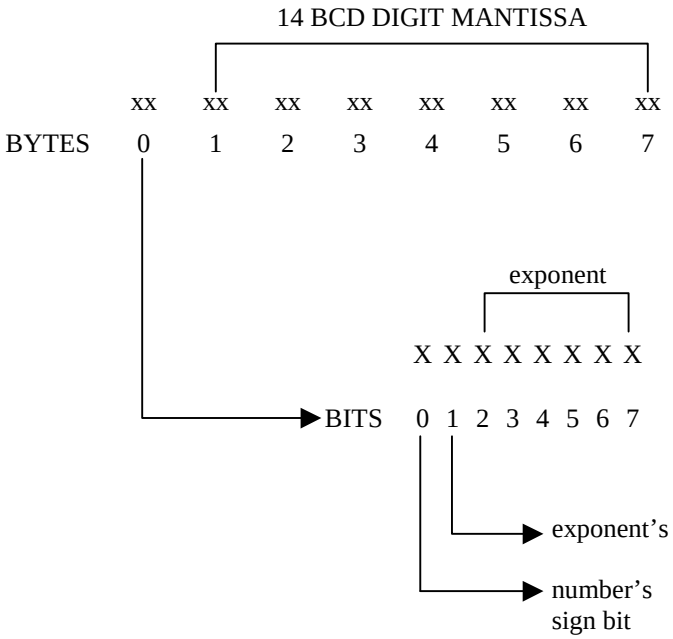

Figure 6-2 Real Number Storage

•INTEGERS are stored in two bytes of memory space with the low-order byte first, as shown in the following diagram. Integers are represented as 16-bit two's complement binary numbers. Integer values are limited to plus or minus 32767.

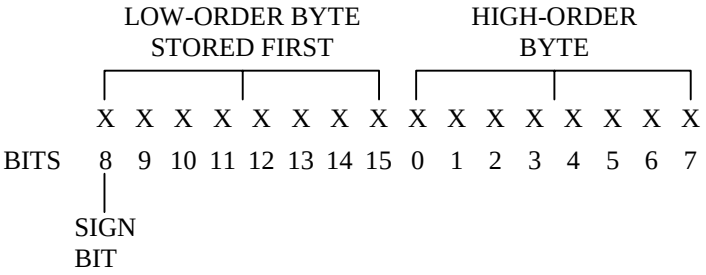

Figure 6-3 Integer Storage

- STRINGS are stored as a sequential list of ASCII representations. The length of a string is stored in the first byte, followed by the actual ASCII values. The maximum number of characters in a string is 255
- ARRAYS, both numeric and string, are allocated space in the Free Storage Area as required. Eight bytes are reserved for each element of an array containing real numbers and two bytes for each element of an integer array. String arrays are allocated three bytes for each entry. A dope vector precedes each array. The dope vector consists of one byte to indicate the number of dimensions, and two additional bytes per dimension to indicate an offset value.

6.3 Assembly Language Interface

CBASIC supports statements and functions that enable assembly language routines to be executed from CBASIC programs.

- CALL statement
- SAVEMEM statement
- PEEK function
- POKE statement
- SADD function
- VARPTR function

The assembler linkage process for both 8-bit and 16-bit environments consists of four steps.

1. Write and debug an assembly language routine.
2. Configure the routine for use with CBASIC. Create a COM file for 8080 routines, or a .CMD file for 8086 routines.
3. Load the routine using the proper SAVEMEM parameters.
4. Write a CBASIC program to CALL the assembly routine.

There are differences in the architecture of 8-bit and 16-bit microprocessors. Therefore, procedures for linking to assembly language routines differ in the two systems. The following demonstration programs for both versions link to assembly routines to perform three simple operations.

- Input a character string from the keyboard.
- Pass the address of the string to an assembler routine.
- Print the string using a BDOS function call.

6.4 CBASIC 8-bit (8080) Demonstration Program

Enter the following 8080 assembly language program into a file named 8080.ASM.

```
      ORG   0E786H
CON$OUT     EQU   2
PRINT$      EQU   9
BDOS        EQU   5
;
      JMP   START
PARAM DW    0     ; ADDRESS IS POKED HERE

START:
      LHLD  PARAM ; POINTER TO ADDRESS OF VARIABLE
      MOV   E,M
      INX   H
      MOV   D,M   ;MOVE ADDRESS TO DE
      XCHG        ;ADDRESS OF VARIABLE IN HL
      MOV   A,M   ;GET LENGTH OF STRING
      INX   H     ;POINT TO FIRST CHARACTER
;
PRINT$LOOP:
      MOV   E,M   ;GET CHARACTER
      INX   H
      MVI   C,CON$OUT   ;SET UP FOR BDOS CALL
      PUSH  H           ;SAVE H REGISTER
      PUSH  PSW         ;SAVE A REGISTER
      CALL  BDOS
      POP   PSW
      POP   H           ;RESTORE REGISTERS
      DCR   A           ;DECREMENT COUNTER
      JNZ   PRINT$LOOP  ;LOOP TILL OUT OF CHARACTERS
      LXI   D,CR$LF     ;ADDRESS OF CR$LF SEQUENCE
      MVI   C,PRINT$    ;PRINT STRING
      CALL  BDOS
      RET               ;RETURN TO CBASIC
CR$LF DB    0DH,0AH,'$'

      END
```

The ORG address in 8080.ASM might not be correct for your particular system, because the address at the top of the Transient Program Area (TPA) varies for different sized configurations of CP/M. The SAVEMEM statement in a CBASIC program reserves the specified amount of memory space for an assembler routine, then loads the routine in that space at the top of the TPA. The SAVEMEM statement calculates the load address by subtracting the size of the routine from the address at the top of the TPA. SAVEMEM does not reveal the load address to you. You must calculate it yourself and specify it in the ORG statement in the assembler routine and in the CALL statement in the CBASIC program. Therefore, you must determine the top of the TPA in your system. CP/M stores that address at 06H and 07H. Use the following CBASIC program to determine the top of your TPA and calculate the load address for your system.

```
REM PROGRAM TO DETERMINE TOP OF TPA
INPUT "ENTER SIZE OF ASSEMBLY ROUTINE. ROUND UP TO\
      NEAREST 128 BYTE INCREMENT. ";FILE.SZE
REM CALCULATE TOP OF TPA AND SUBTRACT FILE SIZE
LOAD.ADDR=PEEK(7)*256+PEEK(6)+65536-F1LE.SZE
PRINT: PRINT LOAD.ADDR;" = LOAD ADDRESS...CONVERT TO HEX."
END
```

If the load address for your system differs from the ORG address in 8080.ASM, edit the correct address into 8080.ASM and assemble the file to create 8080.HEX. If you are running under CP/M Plus, use the MAC™ program to create 8080.HEX.

Note: If you are running under CP/M Plus™ convert the assembly routine into an executable file of type .COM using the CP/M Plus HEXCOM utility. HEXCOM generates executable command files from HEX input files. Name the new executable file 8080.COM.

Note: If you are running under CP/M version 2.1 or earlier, convert the assembly routine into an object code file of type .COM using the CP/M Dynamic Debugging Tool, DDT™. DDT loads a HEX file and converts it to binary format. The R command reads the file into memory starting at the ORG address. However, to SAVE the memory image as a .COM file, the R command must contain am offset value to load the file at l00H. Calculate the offset value using DDT.

```
A>DDT
 DDT VERS X.X
-H100 E786  ;SPECIFY 100, THEN THE LOAD ADDRESS
E886 197A
```

The second number displayed by DDT (197A in this case) is the offset value. Use the I command to specify the filename.

```
-I8080.HEX
```

Use the R command with the offset value to read the HEX file in at 100H.

```
-R197A

NEXT PC
O11E 0000
```

The HEX file is converted to binary form. Now, SAVE the routine as a .COM file.

```
-C
A>SAVE 1 8080.COM
A>
```

When calling or passing parameters to the assembler routine, addresses are absolute memory locations. The CALL statement to start execution of 8080.COM is CALL 0E786H. To pass a parameter, information must be inserted into the routine using the POKE statement. The VARPTR function is used to obtain a pointer to the address of the string variable. The address is converted into a high and low byte, then POKEd directly into the routine. The routine can access the string from CBASIC's Free Storage Area and print it on the console. The information in the string variable can be changed but the length must remain constant. The following CBASIC program calls the 8080.COM file.

```
REM RESERVE 128 HYTES AND LOAD ROUTINE
SAVEMEM 128, "8080.COM"
      PROG.BASE=(PEEK(7)*256+PEEK(6)-128)
      PARAM.OFFSET=PROG.BASE+3
INPUT "ENTER STRING TO PRINT..."; LINE$
WHILE (LINE$ <> "DONE")
      ADDR=VARPTR(LINE$)
      BYTE2=INT(ADDR/256)
      BYTE1=ADDR-(BYTE2*256)
      POKE PARAM.OFFSET,BYTE1 : POKE PARAM.OFFSET+1,BYTE2
CALL 0E786H
      INPUT "ENTER STRING TO PRINT..."; LINE$
WEND
END
```

6.5 CBASIC 16-bit (8086) Demonstration Program

Enter the following assembly language routine into a file named 8086.A86.

```
      CSEG
CON_OUT           EQU   2
PRINT_STRING      EQU   9
BDOS              EQU   224
      ORG   100H
      JMP   START
PARAM DW    0           ;POKE ADDRESS HERE
START:
;
;SAVE REGISTERS FROM CBASIC,
;DS POINTS TO CBA6IC DATA AREA
;
      PUSH  SI
      PUSH  BP
      MOV   SI,CS:PARAM       ;GET POINTER
                              ;TO ADDRESS
                              ;OF VARIABLE

      MOV   SI,[SI]           ;GET ADDRESS
                              ;OF VARIABLE

      XOR   AX,AX

      LODSB                   ;GET STRING
                              ;LENGTH

      MOV   CX,AX
```

```
PRINT_LOOP:
      PUSH  CX
      LODSB
                              ;GET CHARACTER
      PUSH  SI
      MOV   DL,AL             ;SET UP FOR
                              ;BDOS CALL
      MOV   CL,CON_OUT
      INT   BDOS
      POP   SI
      POP   CX
      LOOP  PRINT_LOOP
;
      MOV   DX,CS:OFFSET CR_LF
;
;DS MUST POINT TO PROGRAM SEGMENT ON BDOS
;CALL ELSE, DX REFERENCES AN AREA IN CBASIC
;DATA AREA
;
      PUSH  DS          ;SAVE DS FOR CBASIC
      MOV   AX,CS
      MOV   DS,AX       ;NOW DS POINTS TO
                        ;SEGMENT FOR CR_LF
;
      MOV   CL,PRINT_STRING
      INT   BDOS
;
;RESTORE REGISTERS HERE
;
      POP   DS          ;DS POINTS TO CBASIC
                        ;DATA AREA
      POP   BP
      POP   SI
;
;RETURN TO CBASIC WITH FAR RETURN
;
      RETF

CR_LF DB    0DH,0AH,"$"

      END
```

Use GENCMD to convert 8086.H86 to a .CMD file. Now, the assembly routine is ready to be called from the following CBASIC-86 program.

```
SAVEMEM 512, “8086.CMD”
REM CALCULATE TOP OF TPA, SUBTRACT FILE SIZE
REM AND ROUND DOWN TO 16 BYTE BOUNDARY
      PROG.BASE=(PEEK(7)*256+PEEK(6)-512) AND 0FFF0H REM
ADD OFFSET WITHIN ROUTINE
      PARAM.OFFSET=( PROG.BASE+103H)
      INPUT “ENTER STRING TO PRINT...”; LINE$
      WHILE (LINE$<> “DONE”)
REM GET POINTER TO STRING VARIABLE
      ADDR=VARPTR ( LINE$)
REM CALCULATE HIGH BYTE OF ADDRESS
      BYTE2=INT(ADDR/256)
REM CALCULATE LOW BYTE OF ADDRESS
      BYTE1=ADDR-(BYTE2*256)
REM POKE BYTES IN LOW, HIGH ORDER
      POKE PARAM.OFFSET, BYTE1: \
      POKE PARAM.OFFSET + 1,BYTE2
REM CALL ROUTINE, START AT BEGINNING
      CALL 100H
      INPUT “ENTER STRING TO PRINT..,”;LINE$ WEND
END
```

In CBASIC-86, the CALL address is an offset from the beginning of the routine code segment. Specify the offset in the routine. When passing parameters with the POKE and PEEK statements, the address used is an offset from the base of the CBASIC-86 data segment. You must determine the address of the routine in the CP/M-86 TPA.

To calculate the program base offset value, use the address at 06H and 07H. Subtract the size of the file and round down to a 16 byte paragraph boundary. Once the program base is determined, add the offset value for the data area that you want to access.

When control is transferred to the assembly routine, the following rules apply:

- The DS, SS, BP and SI registers must be saved, then restored prior to returning to the CBASIC-86 program.
- If a routine must reference data in its data segment, the DS register must be initialized in the routine.
- Return to the CBASIC-86 program with a far return.

VARPTR generates a pointer to access a string in the CBASIC-86 string space. The information in the string can be changed but the length must remain constant. Before passing an address to the routine, convert the address into two bytes transferable with the POKE statement.

End of Section 6

Section 7
Compiling and Running CBASIC Programs

7.1 Compiler Directives

Compiler directives are special commands that control the processing action of the compiler. All compiler directives are preceded by a percent sign, % that must be entered in column one. CBASLC supports six Compiler directives.

- %LLST
- %NOLIST
- %PAGE
- %EJECT
- %INCLUDE
- %CHAIN

The Compiler ignores characters following a directive on the same line.

7.2 Listing Control

Four compiler directives control the listing of Compiler messages.

- The %LIST directive turns the Compiler listing on. %LIST can be placed anywhere in a program, and can be used any number of times in conjunction with %NOLIST. This allows selected portions of a program to be listed. Both directives can affect listings to a console, printer, or disk file.
- The %NOLIST directive turns the Compiler listing off.
- The %PAGE directive sets the page length that is output to a printer. A constant enclosed in parentheses must be specified following the directive. For example, %PAGE(45) sets the page length to 45 lines. Initially, the page length is set at 64 lines. The constant must be a positive integer value greater than zero. Any number of %PAGE directives can be used in a program.
- The %EJECT directive positions the printer to list Compiler messages at the top of the next blank page of paper. The directive sends a form feed to the printer.

7.3 %INCLUDE Directive

The %INCLUDE directive allows a specified CBASIC program to be compiled and executed from within another CBASIC program. %INCLUDE directives can specify only one file on a line, and directives cannot reference themselves. However, %INCLUDE directives can be nested up to six deep. The filename specified can contain a drive reference as in the following example. The Compiler assumes a .BAS filetype.

```
%INCLUDE B:PROGRAM
```

The example includes the file PROGRAM.BAS on drive B into the compilation and execution of the original program.

Because the files incorporated with %INCLUDE directives are of filetype .BAS, they can be compiled separately. It is easier to debug large programs if they are composed of small, individually tested routines. Routines to be included by the %INCLUDE directive must not contain an END statement.

The %INCLUDE directive allows you to build a library of common routines, thus reducing programming time. System standards, such as I/O port assignments, can be put in included routines. If the programs are moved from one system to another, the INCLUDE routine is changed, and the programs must be recompiled.

Commonly used procedures, such as searches, validation routines, or input routines, are candidates for INCLUDE files. All file access commands, such as READ, PRINT, or OPEN, can be set up as separate INCLUDE files if certain files are accessed frequently.

You may notice that a program segment can be compiled without errors when compiled separately, but when combined with other routines can cause Compiler errors. These errors are usually quite obvious. They often result from using the same line number in more than one module.

7.4 %CHAIN Directive

The %CHAIN directive determines the maximum size of the constant, code, data, and variable areas to be used for a series of chained programs. This ensures that a chained program does not overwrite a portion of the data area passed by a previously

executed program. The four areas correspond to four constants specified after the %CHAIN directive. The four constants are separated by commas, as in the following example.

```
%CHAIN 10,708,0,8
```

The Compiler sets each of the four areas to the values specified in the %CHAIN directive. Each constant must be an unsigned positive integer.

- The first constant is the size of the area reserved for real constants.
- The second constant is the size of the code area.
- The third constant is the area that stores values from data statements.
- The fourth constant is the size of the area that stores variables.

The constants can be expressed as hexadecimal numbers by appending an H to the number. Areas greater than 32,767 must be written as hexadecimal values.

The values used in the %CHAIN directive are determined by compiling each of the programs to be chained and using the largest value of each area. The Compiler lists the size of each area at the end of a compilation. For example, if three programs are to be chained and the CODE SIZE for the programs are 789, 1578, and 4917 bytes, the second constant in the %CHAIN directive is 4917.

The %CHAIN directive is only required in the main or first program executed.

7.5 CBASIC Compile-time Toggles

Enter a Compiler command line using the following syntax.

CBAS *filename* [*disk ref*] [$*toggle* {*toggle*}]

The .INT file is written to the drive specified in the disk ref parameter. If you do not specify a disk ref, the .INT file is written to the drive containing the source file.

The following examples show use of CBASIC Compiler toggles.

```
CBAS ACCOUNT3 $BGF

B:CBAS A:COMPARE $GEC

CBAS PAYROLL $B

CBAS B:VALIDATE $E
```

Compiler toggles are a series of switches that can be set when compiling a program. The toggles are set by typing a $ followed by the letter designations of the desired toggles, starting one space or more after the program name. Toggles are only set for the Compiler. CBASIC supports six compile-time toggles.

Table 7-1 Compile-time Toggles

Toggle	Function
Toggle B	Suppresses the listing of the program on the console during compilation. If an error is detected, the error message is printed even if toggle B is set. Toggle B does not affect listing to the printer (toggle F) or disk file (toggle G). Initially, toggle B is off.
Toggle C	Suppresses the generation of an INT file. Because the first compilation of a large program is likely to have errors, this toggle provides an initial syntax check without the overhead of writing the intermediate file. Initially, toggle C is off.
Toggle D	Suppresses translation of lower-case letters to upper-case. For example, if toggle D is on, AMOUNT does not refer to the same variable as amount. If toggle D is set, all keywords must be capitalized. Initially, toggle D is off.
Toggle E	Useful when debugging programs. If this toggle is set, it causes the run-time program to accompany any error messages with the CBASIC line number where the error occurred. Toggle E increases the size of the resultant INT file and, therefore, should not be used with debugged programs. Toggle E must be set for the TRACE option to be in effect. Initially, toggle E is off.

Table 7-1 (continued)

Toggle	Function
Toggle F	Causes the compiled output listing to be printed on the list device and the console. This provides a hard copy of the compiled program. Even if toggle B is set, a complete listing is provided if toggle F is set. Each page of the listing has a title and numbered pages. Form-feeds are used to advance to the top of a page. Initially, toggle F is off.
Toggle G	Causes the compiled output listing to be written to a disk file. The file containing the compiled listing has the same name as the source file, and a filetype of LST. If toggles G and B are specified, only errors are output at the console, but a disk file of the complete program is produced.

Usually the disk listing is placed on the same drive as the source file. The operator can select another drive by specifying the desired drive, enclosed in parentheses, following toggle G as shown below:

```
CBAS B:TAX $G(A:)
```

Initially, toggle G is off.

7.6 Compiler Output

CBASIC does not require each statement of a program to be assigned a statement number. The only statements that must be given a statement number are those that have control passed to them by the GOTO, GOSUB, ON, or IF statements. During compilation, CBASIC assigns a sequential number to each line independent of the

statement number you used. The CBASIC assigned line number is the one referred to in error messages (if toggle E is specified) and when using the TRACE option. The line number takes one of three forms,

n:

or,

n*

or,

n=

where n is the number assigned. Usually, the colon follows the number. The * is used when the statement contains a user-assigned statement number that is not referenced anywhere in the program. The = is printed when the statement is read in from a disk file with a %INCLUDE directive.

For example:

```
1:    print “start”
2:    name$=“FRED”
3*    10 gosub 40       rem print name
4:    stop
5:
6:%include printrtn     rem rtn to print
7= 40 rem-----rtn to print----------
8=    print name$
9=    return
10:   END
```

In the preceding example, statement 3 has an * because the 10 is not referenced anywhere in the program. This can be useful during debugging, or to help understand large programs written in other Basic dialects. When all un-referenced line numbers are removed, it is easier to see the logic of the program.

When an error is detected, the Compiler prints a two-letter error code, the line number where the error occurred, and the position of the error relative to the beginning of the source line. The position assumes tab characters have been expanded.

7.7 TRACE Option

The TRACE option is used for run-time debugging. It prints the line number of each statement as it is executed. The output is directed to the console even when a LPRINTER statement is in effect. The line number printed is the number the Compiler assigned to each statement. The TRACE option syntax is as follows:

CRUN *filename* [TRACE [*1n1* [,*1n2*]]]

Consider the following program:

```
AMOUNT = 12.13
TIME = 45.0
PRINT TIME * AMOUNT
```

If the preceding program is compiled using the following command,

```
CBAS TEST $E
```

and then executed with the TRACE option,

```
CRUN TEST TRACE 1,3
```

the following output is produced:

```
AT LINE 0001
AT LINE 0002
AT LINE 0003
 545.85
```

The TRACE option functions only if the toggle E is set during compilation of the program.

The first number,1n1, is specifying thee line number where the trace begins. The second number, 1n2, specifies where the trace is to stop. If no line numbers are included in the command, the entire program is traced; if only the first line number is present, tracing starts at this line number and continues for all line numbers greater than the first number 1n1

7.8 Cross-Reference Lister

Besides a Compiler and an Interpreter, a Cross-reference Lister is supplied with CBASIC. The XREF file produces a disk file containing an alphabetized list of all identifiers used in a CBASIC program. The identifier usage (function, parameter, or global) is provided, and a list of each line where that identifier is used.

The listing places all functions first, parameters and local variables associated with a function immediately follow. The functions are listed alphabetically. The output is usually directed to the same disk as the source file. The file created has the same name as the CBASIC source file and is of type XREF. The standard output is 132 columns wide.

The following command is used to invoke XREF:

XREF *filename* [disk ref] [$*toggles*] [‘*title*’]

The filename must be a CBASIC source program with a filetype of BAS. The disk reference is optional and specifies on which disk to place the cross-reference file. If the disk reference is not present, the listing is placed on the same drive as the source. It is specified as A, B, etc.

For example,

```
XREF PAYROLL A:
```

places the cross-reference listing for PAYROLL.BAS on drive A. At least one blank must separate the filename and the disk reference.

Toggles can alter the standard output of XREF. A, B, C, D, E, F, G, and H are valid toggles. They can be lower- or upper-case letters. At least one blank must separate the $ from the portion of the command line to the left. The toggles follow the $. The Cross-reference Lister ignores any other characters that follow the $, and precede the title field or end of the command line.

The following is a list of the Cross-reference Lister toggles and their functions:

Table 7-2 Cross-reference Lister Toggle Functions

Toggle	Function
A	Causes a listing to be output to the list device and to a disk file.
B	Suppresses output to the disk. If only the B toggle is specified, no output is produced.
C	Suppresses the output to the disk and permits output to the list device. The C toggle has the same effect as specifying both the A and B toggles.
D	Causes the output to be produced 80 columns wide instead of using 132 columns.
E	Produces output with only the identifiers and their usage. No line numbers are printed. The E toggle helps document a program. You write the use of each identifier on the listing provided by XREF. The file created by XREF is edited and made into a large remark with comments pertaining to each variable name. By including this file with the source program, more documentation is provided.
F	Allows you to change the default page length of 60 lines per page. The desired number of lines per page is enclosed in parentheses and must follow the F toggle. Embedded blanks are not allowed. Form-feed characters position the printer, and are also placed in disk files.
G	Suppresses printing of the heading lines and suppresses all form-feeds. This toggle is used when building a disk file to be printed by a user utility.
H	Suppresses translation of lower-case letters to upper-case. This allows using XREF with programs compiled with compiler toggle D.

The following command:

```
XREF ACCT$REC B: $EAF(40)
```

creates a disk file on drive B and a listing on the list device of all the identifiers and their usage. No line numbers are provided. Pages are limited to 40 lines.

The optional title field must be the last field in the command line. All characters following the first apostrophe on the command line up to the second apostrophe, or until the end of the command line, become the title. The title is printed on the heading line of each page of output. The title is truncated to 30 characters if the listing is 132 columns wide, or to 20 characters if the D toggle is specified.

The following command demonstrates the title field:

```
XREF NAMES B: $AD ‘version 2: 1 AUG 78’
```

End of Section 7

Appendix A
Compiler Error Messages

The compiler prints the following messages when a file system error or memory space error occurs. In each case, control returns to the operating system.

Table A-1 File System and Memory Space Errors

Error	*Meaning*
NO SOURCE FILE: filename.BAS	The compiler cannot locate a source file on the specified disk. This file was used in either the CBAS2 command or a %INCLUDE directive.
OUT OF DISK SPACE	The compiler has run out of disk space while attempting to write either the TNT file or the LST file.
OUT OF DIRECTORY SPACE	The compiler has run out of directory entries while attempting to create or extend either the INT file or the LST file.
DISK ERROR	A disk error occurred while trying to read or write to a disk file. This message can vary slightly in form depending on the operating system used. See the CP/M documentation for the exact meaning of this message.
PROGRAM CONTAINS n UNMATCHED FOR STATEMENT(S)	There are n FOR statements for which a NEXT cannot be found.
PROGRAM CONTAINS n UNMATCHED WHILE STATEMENT(S)	There are n WHILE statements for which a WEND cannot be found.

Error	*Meaning*
PROGRAM CONTAINS 1 UNMATCHED DEF STATEMENT	A multiple line function was not terminated with a FEND statement. This causes other errors in the program.
WARNING INVALID CHARACTER IGNORED	The previous line contained an invalid character. The compiler ignores the character; a question mark is printed in its place.
INCLUDE NESTING TOO DEEP NEAR LINE n	An INCLUDE statement near line n in the source program exceeds the maximum level of nesting of INCLUDE files.

Other errors detected during compilation cause a two-letter error code to be printed with the line number and position of the error. The error message usually follows the line where the error occurred.

Table A-2 Compilation Error Codes

Code	*Error*
BF	A branch into a multiple line function from outside the function was attempted.
BN	An invalid numeric constant was encountered.
CF	A COMMON statement must be in the first line.
CI	An invalid filename was detected in a %INCLUDE directive. The filename cannot contain a ?, *, or : (except as part of a disk reference where a colon can be the second character of the name).
CS	A COMMON statement, that was not the first statement in a program, was detected. Only a compiler directive such as % CHAIN, a REMARK statement, or blank lines can precede a COMMON statement.

Code	*Error*
CV	An improper definition of a subscripted variable in a COMMON statement was detected. The subscript count is possibly not a constant, or there is more than one constant. Only one constant can appear in parentheses. It specifies the number of subscripts in the defined array.
DL	The same line number was used on two different lines. Other compiler errors can cause a DL error message to be printed even if duplicate line numbers do not exist. Defining functions before use and, sometimes, if the DIM statement does not precede all references to an array, results in a DL error.
DP	A variable dimensioned by a DIM statement was previously defined. It either appears in another DIM statement or was used as a simple variable.
FA	A function name appears on the left side of an assignment statement but is not within that function. In other words, the only function name that can appear to the left of an equal sign is the name of the function currently compiled.
FD	The same function name is used in a second DEF statement.
FE	A mixed mode expression exists in a FOR statement that the compiler cannot correct. Probably the expression following the TO is of a different type than the index.
FI	An expression is a subscripted numeric variable being used as a FOR loop index.
FN	A function reference contains an incorrect number of parameters.
FP	A function reference parameter type does not match the parameter type used in the function's DEF statement.
FU	A function was referenced before it was defined, or the function was never defined.
IE	An expression immediately following an IF statement evaluates to type string. Only type numeric is permitted.

Code	*Error*
IF	A variable used in a FILE statement is of type numeric where type string is required.
IP	An input prompt string was not surrounded by quotes.
IS	A subscripted variable was referenced before dimensioned.
IT	An invalid compiler directive was encountered. A parameter required by the directive can be out of range or missing, or the directive was misspelled.
IU	A variable defined as an array in a DEF statement is used without subscripts.
MC	The same variable is defined more than once in a COMMON statement. Each variable can appear in only one COMMON statement.
MF	An expression evaluates to type string when an expression of type numeric is required.
MM	An invalid mixed mode was detected. Probably variables of type string and type numeric were combined in the same expression.
MS	A numeric expression was used where a string expression was required.
ND	A FEND statement was encountered without a corresponding DEF statement. This error could be the result of an improper DEF statement.
NI	A variable referenced by a NEXT statement does not match the variable referenced by the associated FOR statement.
NU	A NEXT statement occurs without an associated FOR statement.
OF	A branch out of a multiple line function from inside the function was attempted.

Code	*Error*
00	More than 40 ON statements were used in the program. CBASIC has an arbitrary limit of 40 ON statements in a single program. Notify Digital Research if this limit causes problems.
PM	A DEF statement appeared within a multiple line function. Functions cannot be nested.
RF	A multiple line function cannot call itself.
SD	A second SAVEMEM statement was encountered. A program can have only one SAVEMEM statement.
SE	The source line contained a syntax error. This means that a statement is not properly formed or a keyword is misspelled.
SF	A SAVEMEM statement uses an expression of type numeric to specify the file to be loaded. The expression must be a string. Possibly, the quotation marks were left off a string constant.
SN	A subscripted variable contains an incorrect number of subscripts, or a variable in a DIM statement was previously used with a different number of dimensions.
SO	The statement is too complex to compile; simplify it. Consider making the expression into two or more expressions. Please send Digital Research a copy of the source statement.
TO	Symbol table overflow has occurred. This means that the program is too large for the system being used. The program must be simplified or the amount of available memory increased. Smaller variable names reduce the amount of symbol table space used. Please inform Digital Research if programs generate this error.
UL	A line number that does not exist was referenced.
US	A string was terminated by a carriage return rather than by quotes.

Code	*Error*
VO	Variable names are too long for one statement. This should not usually occur. If it does, please send a copy of the source statement to Digital Research. Reducing the length of variable names and reducing the complexity of the expression within the statement can eliminate the error.
WE	The expression immediately following a WHILE statement is not numeric.
WN	WHILE statements are nested to a depth greater than 12. CBASIC has an arbitrary limit of 12 for nesting WHILE statements.
WU	A WEND statement occurred without an associated WHILE statement.

End of Appendix A

Appendix B
Run-time Error Messages

The following warning messages might be printed during execution of a CBASIC program.

Table B-1 CBASIC Warning Messages

Error	*Meaning*
NO INTERMEDIATE FILE:*filename*	A filename was not specified with the CRUN2 command, or no file of type INT with the specified filename was found on the specified disk.
IMPROPER INPUT - REENTER	This message occurs when the fields entered from the console do not match the fields specified in the INPUT statement. This occurs when field types do not match or the number of fields entered differs from the number of fields specified. Following this message, all values required by the INPUT statement must be reentered.

Run-time errors cause a two-letter code to be printed. If the code is preceded by the word WARNING, execution continues. If the code is preceded by thee word ERROR, execution terminates. If an error occurs with a code consisting of an * followed by a letter such as * R, the CBASIC run-time package has failed. Please notify Digital Research of the circumstances under which the error occurred.

The following table contains valid CBASIC warning codes.

Table B-2 CBASIC Warning Codes

Code	*Error*
DZ	A number was divided by zero. The result is set to the largest valid CBASIC number.
FL	A field length greater than 255 bytes was encountered during a READ LINE statement. The first 255 characters of the record are retained; the other characters are ignored.
LN	The argument given in the LOG function was zero or negative. The value of the argument is returned.
NE	A negative number was specified before the raise to a power operator. The absolute value of the parameter is used in the calculation. When using real variables, a positive number can be raised to a negative power, but a negative number cannot be raised to a power.
OF	A calculation using real variables produced an overflow. The result is set to the largest valid CBASIC real number. Overflow is not detected with integer arithmetic.
SQ	A negative number was specified in the SQR function. The absolute value is used.

The following table contains valid CBASIC error codes.

Table B-3 CBASIC Error Codes

Code	*Error*
AC	The argument in an ASC function is a null string.
AE	An attempt was made to access an array element before the array DIM statement was executed.
BN	The value following the BUFF option in an OPEN or CREATE statement is less than one or greater than 52.

Code	*Error*
CC	A chained program's code area is larger than the main program's code area. Use a %CHAIN directive in the main program to adjust the size of the code area.
CD	A chained program's data area is larger than the main program's data area. Use a %CHAIN directive in the main program to adjust the size of the data area.
CE	The file being closed cannot be found in the directory. This occurs if the RENAME function has changed the file.
CF	A chained program's constant area is larger than the main program's constant area. Use a %CHAIN directive in the main program to adjust the size of the constant area.
CP	A chained program's variable storage area is larger than the main program's variable storage area. Use a %CHAIN directive in the main program to adjust the size of the variable storage area.
CS	A chained program reserved a different amount of memory, with a SAVEMEM statement, than the main program.
CU	A CLOSE statement specifies a file identification number that is not active.
DF	An OPEN or CREATE statement uses a file identification number that is already used.
DU	A DELETE statement specifies a file identification number that is not active.
DW	The operating system reports that there is no disk or directory space available for the file being written to, and no IF END statement is in effect for the file identification number.
EF	An attempt is made to read past the end of a file, and no IF END statement is in effect for the file identification number.

Code	*Error*
ER	An attempt was made to write a record of length greater than the maximum record size specified in the OPEN, CREATE, or FILE statement for this file.
FR	An attempt was made to rename a file to an existing filename.
FU	An attempt was made to access a file that was not open.
IF	A filename was invalid. Most likely, an invalid character was found in the filename. A colon can never appear embedded in the name proper. ? and * can only appear in ambiguous filenames. This error also results if the filename was a null string.
IR	A record number of zero was specified in a READ or PRINT statement.
IV	An attempt was made to execute an INT file created by a version one compiler. To use CRUN2, a program must be recompiled using the version two compiler, CBAS2. This error also results from attempting to execute an empty INT file.
IX	A FEND statement was encountered before executing a RETURN statement. All multiple line functions must exit with a RETURN statement.
ME	The operating system reports an error during an attempt to create or extend a file. Usually this means the disk directory is full.
MP	The third parameter in a MATCH function was zero or negative.
NC	The source program contains a real constant outside the range of CBASIC real numbers.
NF	A file identification number is less than one or greater than 20, or a FILE statement was executed when 20 files were already active.
NM	There was insufficient memory to load the program.

Code	*Error*
NN	An attempt to print a numeric expression with a PRINT USING statement fails because there is not a numeric field in the USING string.
NS	An attempt to print a string expression with a PRINT USING statement fails because there is not a string field in the USING string.
OD	A READ statement was executed, but there are no DATA statements in the program, or all data items in all DATA statements were already read.
OE	An attempt was made to OPEN a file that does not exist and for which no IF END statement was in effect.
OI	The expression specified in an ON ... GOSUB or an ON ... GOTO statement evaluated to a number less than one or greater than the number of line numbers contained in the statement.
OM	The program ran out of dynamically allocated memory during execution. Space can be conserved by closing files when no longer needed, and by setting strings to a null string when no longer required. Also, by not using DATA statements, but reading the constant information from a file, space is saved. Large arrays can be dimensioned with smaller subscripts when the array is no longer required.
QE	An attempt was made to print a string containing a quotation mark to a file. Quotation marks can only be written to files when using the PRINT USING option of the PRINT statement.
RB	Random access was attempted to a file activated with the BUFF option specifying more than one buffer.
RE	An attempt was made to read past the end of a record in a fixed file.
RF	A recursive function call was attempted; CBASIC does not support recursion.
RG	A RETURN statement occurred for which there was no GOSUB statement.

Code	*Error*
RU	A random read or print was attempted to a stream file.
SB	An array subscript was used which exceeds the boundaries for which the array was defined.
SL	A concatenation operation resulted in a string greater than the maximum allowed string length.
SO	The file specified in a SAVEMEM statement cannot be located on the referenced disk. The expression specifying the filename must include the filetype if one is present. A filetype of COM is not forced.
SS	The second parameter of a MID$ function was zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ function was negative.
TL	A TAB statement contains a parameter less than one.
UN	A PRINT USING statement was executed with a null edit string, or an escape character is the last character in an edit string.
WR	An attempt was made to write to a stream file after it was read, but before it was read to the end of the file.

End of Appendix B

Appendix C
CBASIC Key Words

ABS	ELSE	LE	PRINT	STR$
AND	END	LEFT$	RANDOMIZE	SUB
AS	EQ	LEN	READ	TAB
ASC	EXP	LET	RECL	TAN
ATN	FEND	LINE	RECS	THEN
BUFF	FILE	LOG	REM	TO
CALL	FLOAT	LPRINTER	REMARK	UCASE$
CHAIN	FN*	LT	RENAME	USING
CHR$	FOR	MATCH	RESTORE	VAL
CLOSE	FRE	MIDS	RETURN	VARPTR
COMMANDS	GE	NE	RIGHTS	WEND
COMMON	GO	NEXT	RND	WHILE
CONCHAR%	GOSUB	NOT	SADD	WIDTH
CONSOLE	GOTO	ON	SAVEMEM	XOR
CONSTAT%	GT	OPEN	SGN	%CHAIN
COS	IF	OR	SIN	%EJECT
CREATE	INITIALIZE	OUT	SIZE	%INCLUDE

DATA	INP	PEEK	SQR	%LIST
DEF	INPUT	POKE	STEP	%NOLIST
DELETE	INT	POS	STOP	%PAGE
DIM	INT%			

*For FN, see user-defined functions.

End of Appendix C

Appendix D
Decimal-ASCII-Hex Table

Table D-1 Conversion Table

DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX
0	NUL	00	44	,	2C	88	X	58
1	SOH	01	45	-	2D	89	Y	59
2	STX	02	46	.	2E	90	Z	5A
3	ETX	03	47	/	2F	91	[	5B
4	EOT	04	48	0	30	92	\	5C
5	ENQ	05	49	1	31	93	]	5D
6	ACK	06	50	2	32	94	^	5E
7	BEL	07	51	3	33	95		5F
8	BS	08	52	4	34	96	`	60
9	HT	09	53	5	35	97	a	61
10	LF	0A	54	6	36	98	b	62
11	VT	0B	55	7	37	99	c	63
12	FF	0C	56	8	38	100	d	64
13	CR	0D	57	9	39	101	e	65
14	SO	0E	58	:	3A	102	f	66
15	SI	0F	59	;	3B	103	g	67
16	DLE	10	60	<	3C	104	h	68
17	DC1	11	61	=	3D	105	i	69
18	DC2	12	62	>	3E	106	j	6A
19	DC3	13	63	?	3F	107	k	6B
20	DC4	14	64	@	40	108	l	6C
21	NAK	15	65	A	41	109	m	6D
22	SYN	16	66	B	42	110	n	6E
23	ETB	17	67	C	43	111	o	6F
24	CAN	18	68	D	44	112	p	70
25	CR	19	69	E	45	113	q	71
26	SUB	1A	70	F	46	114	r	72
27	ESC	1B	71	G	47	115	s	73
28	FS	1C	72	H	48	116	t	74
29	GS	1D	73	1	49	117	u	75
30	RS	1E	74	J	4A	118	v	76
31	US	1F	75	K	4B	119	w	77
32	SP	20	76	L	4C	120	x	78
33	!	21	77	M	4D	121	y	79
34	"	22	78	N	4E	122	z	7A

DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX
35	#	23	79	O	4F	123	{	7B
36	$	24	80	P	50	124	|	7C
37	%	25	81	Q	51	125	}	7D
38	&	26	82	R	52	126		7E
39	'	27	83	S	53	127	DEL	7F
40	(	28	84	T	54			
41	)	29	85	U	55			
42	*	2A	86	V	56			
43	+	2B	87	W	57			

End of Appendix D

Appendix E
Glossary

address: Location in memory.

ambiguous file specification: File specification that contains either of the Digital Research wildcard characters, ? or *, in the filename or filetype or both. When you replace characters in a file specification with these wildcard characters, you create an ambiguous filespec and can reference more than one file in a single command line.

applications program: Program that needs an operating system to provide an environment in which to execute. Typical applications programs are business accounting packages, word processing, and mailing list programs.

argument: Variable or expression value that is passed to a procedure or function and substituted for the dummy argument in the function. Same as “actual argument” or “calling argument”. Used interchangeably with “parameter .

array: Data type that is itself a collection of individual data items of the same data type. Term used to describe a form of storing and accessing data in memory, visualized as matrices. The number of extents of an array is the number of dimensions of the array. A one dimensional array is essentially a list.

ASCII: Acronym for American Standard Code for Information Interchange. ASCII is a standard code for representation of the numbers, letters, and symbols that appear on most keyboards.

assembler: Language translator that translates assembly language statements into machine code.

assignment statement: Statement that assigns the value of an expression on the righ side of an equal sign to the variable name on the left side of the equal sign.

back-up: Copy of a file or disk made for safe keeping, or the creation of the file o disk.

binary: Base two numbering system containing the two symbols zero and one.

bit: Common contraction for “binary digit”. “Switch” in memory that can be set to on (1) or off (0). Eight bits grouped together comprise a byte.

buffer: Area of memory that temporarily stores data during the transfer of information.

byte: Unit of memory or disk storage containing eight bits.

call: Transfer of control to a computer program subroutine.

chain: Transfer of control from the currently executing program to another named program without returning to the system prompt or invoking the run-time monitor.

code: Sequence of statements of a given language that make up a program.

command: Instruction or request for the operating system or a system program to perform a particular action. Generally, a Digital Research command line consists of a command keyword, a command tail usually specifying a file to be processed, and a carriage return.

common: Variables used by a main program and all programs executed through a chain statement.

compiler: Language translator that translates the text of a high level language into machine code.

compiler directive: Reserved words that modify the action of the compiler.

compiler error: Error detected by the compiler during compilation; usually caused by improper formation of language statement.

compiler toggle: “Switch” to modify the output of the compiler.

concatenate: Join one string to another or one file to another.

concatenation operator: Symbol peculiar to a given language that instructs the compiler to combine two unique data items into one.

console: Primary input/output device. Thee console consists of a listing device such as a screen and a keyboard through which the user communicates with the operating system or the applications program.

constant: String or numeric value that does not change throughout program execution.

control character: Nonprinting character combination that sends a simple command to the operating system or applications program. To enter a Control character, press the Control (CTRL) key on your terminal and strike the character key specified.

control statement: Language statement that transfers control or directs the order of execution of instructions by the processor.

cursor: One-character symbol that can appear anywhere on the video screen. The cursor indicates the position where the next keystroke at the console will have an effect.

data: Information; numbers, figures, names, and so forth.

data base: Large collection of information, usually covering various aspects of related subject matter.

data file: Non-executable file of similar information that generally requires a command file to process it.

data structure: Mechanism, including both storage layout and access rules, by which information can be stored and retrieved within a computer system. Data structures can reside in memory or on secondary storage. System tables such as symbol tables, matrices of numerical data, and data files are examples of data structures.

data type: Class or use of the data; for example, integer, real or string.

debug: Remove errors from a program.

default: Values, parameters or options a given command assumes if not otherwise specified.

delimiter: Special characters or punctuation that separate different items in a command line or language statement.

dimension: Refers to the number of extents of an array. A one dimensional array is essentially a list of the elements of the array. A two dimensional array can be visualized as a matrix of rows and columns of storage space for the elements of the array. A three dimensional array can be thought of as a geometric solid having volume, and so forth.

directory: Portion of a disk that contains entries for each file on the disk. In response to the DIR command, CP/M and MP/M systems display the file specifications stored in the directory.

disk, diskette: Magnetic media used to store information. Programs and data are recorded on the disk in the same way that music is recorded on a cassette tape. The term "diskette" refers to smaller capacity removable floppy diskettes. The term "disk" can refer to a diskette, a removable cartridge disk, or a fixed hard disk.

disk drive: Peripheral device that reads and writes on hard or floppy disks. CP/M and MP/M systems assign a letter to each drive under their control.

drive specification: Alpha character A-P followed by a colon that indicates the CP/ M or MP/M drive reference for the default or specified drive.

dummy argument: Argument used in the definition of a command or language statement (especially a function) that holds a place that will later contain a usable "actual" or "calling" argument that is passed to the function by a calling statement. Same as "formal argument".

editor: Utility program that creates and modifies text files. An editor can be used to create documents or code for computer programs.

element: Individual data item in an array.

executable: Ready to run on the processor. Executable code is a series of instructions that can be carried out on the processor. For example, the computer cannot "execute" names and addresses, but it can execute a program that prints names and addresses on mailing labels.

execute a program: Start a program running. When the program is executing, a process is executing a sequence of instructions.

FCB: File Control Block. Structure used for accessing files on disk. Contains the drive, filename, filetype and other information describing a file to be accessed or created on the disk.

field: Portion of a record; length and type are defined by the programmer. One or more fields comprise a record.

file: Collection of related records containing characters, instructions or data; usually stored on a disk under a unique file specification.

filename: Name assigned to a file. The filename can include 1-8 alpha, numeric and/or some special characters. The filename should tell something about the file.

filetype: Extension to a filename. A filetype is optional, can contain from 0 to 3 alpha, numeric and/or some special characters. The filetype must be separated from the filename by a period. Certain programs require that files to be processed have specific filetypes.

file access: Refers to methods of entering a file to retrieve the information stored in the file.

file specification: Unique file identifier. A Digital Research file specification includes an optional drive specification followed by a colon, a primary filename of 1-8 characters, and an optional period and filetype of 0-3 characters. Some Digital Research operating systems allow an optional semicolon and password of 1-8 characters following the filename or filetype. All alpha and numeric characters and some special characters are allowed in Digital Research file specifications.

fixed: Type of file organization used when data is to be accessed randomly-not in sequential order. Refers generally to the non-varying lengths of the records composing the file.

floating point: Value expressed in decimal notation that can include exponential notation; a real number.

floppy disk: Flexible magnetic disk used to store information. Floppy disks are manufactured in 5¼ and 8 inch diameters.

flowchart: Graphic diagram that uses special symbols to indicate the input, output and flow of control of part or all of a program.

flow of control: Order of the execution of statements within a program.

format: System utility that writes a known pattern of information on a disk so a given hardware configuration can properly support reading and writing on that disk.

formatted printing: Output specifically designed in a certain pattern and achieved through particular coded language statements.

fragmentation: Division of storage area in a way that causes areas to be wasted.

function: Subroutine to which you can pass values and which returns a value. Useful when the same code is required repeatedly, as the program can call the function at any time.

global: Relevant throughout an entire program.

hex file: ASCII-printable representation of a code or data file in hexadecimal notation.

hexadecimal notation: Notation for the base 16 number system using the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F to represent the sixteen digits. Machine code is often converted to hexadecimal notation because it can be more easily understood.

high bound: Upper limit of one dimension of an array.

high-level language: Set of special words and punctuation that allows a programmer to code software without being concerned with internal memory management.

identifier: String of characters used to name elements of a program, such as variable names, reserved words, and user-defined function names. Commonly used synonymously with ”variable name .

include: Call an external file into the code sequence of a program at the point where the include statement is executed.

initialize: Set a disk system or one or more variables to initial values.

I/O: Abbreviation for input/output.

input: Data entered to an executing program, usually from an operator typing at the terminal or by the program reading data from a disk.

instruction: Set of characters that defines an operation.

integer: Positive or negative non-exponential whole number that does not contain a decimal point.

interface: Object that allows two independent systems to communicate with each other, as an interface between the hardware and software in a microcomputer.

intermediate code: Code generated by the syntactical and semantic analyzer portions of a compiler.

interpreter: Computer program that translates and executes each source language statement before translating and executing the next one.

ISAM: Abbreviation for Indexed Sequential Access Method.

key: Particular field of a record on which the processing is performed.

keyword: Reserved word with special meaning for statements or commands.

kilobyte: 1024 bytes denoted as 1K. 32 kilobytes equal 32K. 1024 kilobtyes equal one megabyte, or over one million bytes.

linker: System software module that connects previously assembled or compiled programs or program modules into a unit that can be loaded into memory and executed.

linked list: Data structure in which each element contains a pointer to its predecessor or successor (singly linked list) or both (double linked list).

list device: Device such as a printer onto which data can be listed or printed.

listing: Output file created by the compiler that lists the statements in the source program, the line numbers it has assigned to them, and possibly other optional information.

literal data: Verbatim translation of characters in the code, such as in screen prompts, report titles and column headings.

load: To move code from storage into memory for execution.

local variable: Relevant only within a specific portion of a program, such as within a function.

logged-in: Made known to the operating system, in reference to drives. A drive is logged-in when it is selected by the user or an executing process.

logical: Representation of something such as a console, memory or disk drive that might or might not be the same in its actual physical form. For example, a hard disk can occupy one physical drive, and yet you can divide the available storage on it to appear to the user as if there were several different drives. These apparent drives are the logical drives.

logical device: Reference to an I/O device by the name or number assigned to the physical device.

logical operator: NOT, AND, OR, and XOR.

lower bound: Lower limit of one dimension of an array.

machine code: Output of an assembler or compiler to be executed directly on the target processor.

machine language: Instructions directly executable by the processor.

memory: Storage area within and/or attached to a computer system.

microprocessor: Silicon chip that is the Central Processing Unit (CPU) of the microcomputer system.

mixed mode: Combination of integer and real or numeric and string values in an expression. Mixed string and numeric operations are generally not allowed in high level languages.

mnemonic operator: Alphabetical symbol for algebraic operator: LT, LE, GT, GE, NE, and EQ.

module: Section of software having well-defined input and output that can be tested independently of other software.

multiple-line function: Function composed of a function definition statement and one or more additional statements.

numeric constant: Real or integer quantity that does not vary within the program.

numeric variable: Real or integer identifier to which varying numeric quantities can be assigned during program execution.

null string: A string that contains no character; essentially an empty string. object code: Output of an assembler or compiler that executes on the target processor.

open: System service that informs the operating system of the manner in which a given resource, usually a disk file, is intended to be used.

operating system: Collection of programs that supervises the execution of other programs and the management of computer resources. An operating system provides an orderly input/output environment between the computer and its peripheral devices, enabling user programs to execute safely.

operation: Execution of a piece of code.

option: One of a set of parameters that can be part of a command or language statement. Options are used to modify the output of an executing process.

output: Data that the processor sends to the console, printer, disk, or other storage media.

parameter: Value supplied to a command or language statement that provides additional information for the command or statement. Used interchangeably with ““argument.” An actual parameter is a value that is substituted for a dummy or formal argument in a given procedure or function when it is invoked.

peripheral device: Devices external to the CPU. For example, terminals, printers, and disk drives are common peripheral devices that are not part of the processor, but are used in conjunction with it.

pointer: Data item whose value is the address of a location in memory.

primitive: Most basic or fundamental unit of data such as a single digit or letter.

process: Program that is actually executing, as opposed to being in a static state of storage on disk.

program: Series of specially coded instructions that performs specific tasks when executed on a computer.

prompt: Any characters displayed on the input terminal to help the user decide what the next appropriate action is. A system prompt is a special prompt displayed by the operating system, indicating to the user that it is ready to accept input.

random access: Method of entering a file at any record number, not necessarily the first record in the file.

random access file: File structure in which data can be accessed in a random manner, irrespective of its position in the file.

random number: Number selected at random from a set of numbers.

real number: Numeric value specified with a decimal point; same as “floating point notation.”

record: One or more fields usually containing associated information in numerical or textual form. A file is composed of one or more records and generally stored on disk.

record number: Position of a specific record in a fixed-length file, relative to record number 1. A key by which a specific record in a fixed file is accessed randomly.

recursive: Code that calls itself.

relational operator: Comparison operator. The following set of operators expressed in algebraic or mnemonic symbols: LT, LE, NE, EQ, GT, GE, EQ. A relational operator states a relationship between two expressions.

reserved word: Keyword that has a special meaning to a given language or operating system.

return value: Value returned by a function.

row-major order: Order of assignment of values to array elements in which the first item of thee subscript list indicates the number of ‘“rows” in the array.

run a program: Start a program executing. When a program is running, the microprocessor chip is executing a series of instructions.

run-time error: Error occurring during program execution.

run-time monitor: Program that directly executes the coded instructions generated by a compiler/interpreter.

sequential access: Type of file structure in which data can only be accessed serially, one record at a time. Data can be added only to the end of the file and cannot be deleted. An example of a sequential access media is magnetic tape.

source program: Text file that is an input file for a processing program, such as an editor, text formatter, assembler or compiler.

statement: Defined way of coding an instruction or data definition using specific keywords in a specific format.

storage: Place for keeping data temporarily in memory or permanently on disk.

stream organization: Type of file organization used when data is to be accessed sequentially. Can contain variable length records.

string constant: Literal data, as in a screen prompt, column heading, or title of a report.

string variable: Identifier of type string to which varying strings can be assigned during program execution.

subroutine: Section of code that performs a specific task, is logically separate from the rest of the program, and can be prewritten. A subroutine is invoked by another statement and returns to the place of invocation after executing. Subroutines are useful when the same sequence of code is used more than once in a program.

subscript: Integer expression that specifies the position of an element in an array.

subscript list: Numeric value appended to a variable name that indicates the number of elements in each dimension in the array of that name. Each dimension must have a value in the subscript list indicating the number of elements for which to allocate storage space.

syntax: Rules for structuring statements for an operating system or programming language.

toggle: “Switch” enabled by a special code in the command line that modifies the output of the executing program.

trace: Option used for run-time debugging. The trace option generally lists each line of code as it executes to enable the programmer to note where a problem occurs.

upward-compatible: Term meaning that a program created for the previously released operating system or compiler runs under a later release of the same software program.

user-defined function: Set of statements created and given a function name by the user. The function performs a specific task and is called into action by referencing the function by name.

utility: Tool. Program or module that facilitates certain operations, such as copying, erasing and editing files, or controlling the cursor positioning on the video screen from within a program. Utilities are created for the convenience of programmers and applications operators.

value: Quantity expressed by an integer or real number.

variable: Name to which the program can assign a numerical value or string.

variable length: Usually refers to records, where each record in a file is not necessarily the same length as another.

variable name: Same as variable.

wildcard characters: Special characters, ? and *, that can be included in a Digital Research filename and/or filetype to identify more than one file in a single file specification.

End of Appendix E

Index